Investigations into the suitability of parallel computing architectures for the solution of large sparse matrices using the preconditioned conjugate gradient method

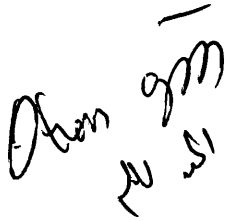
El-Ghajiji, Otman Abubaker

Award date:
1995

Awarding institution:
University of Bath

Investigations into the suitability of parallel computing architectures for the solution of large sparse matrices using the preconditioned conjugate gradient method

Submitted by Mr. Otman Abubaker El-Ghajiji,
B. Sc. (Hons.), M. Sc. (Bath),
For the degree of Ph.D.
of the University of BATH
1995



ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

ACKNOWLEDGMENTS

It was a great pleasure to have Dr. Paul J. Leonard as my supervisor at Bath University. He has been an unfailing source of knowledge, encouragement and support. His guidance was crucial for the development of the ideas for this research and the final form in this thesis.

Special thanks are due to Dr. R. W. Dunn for introducing me to multiprocessor system architectures.

I should also like to thank my colleagues and friends in the School of Electrical Engineering and the university of Bath who have made these past years so enjoyable.

ABSTRACT

This thesis presents investigations into the suitability of parallel computing architectures for the solution of large sparse matrices using the preconditioned conjugate gradient method. The solution is based on the incomplete choleski conjugate gradient algorithm. The preconditioning phase of the algorithm involves a backsubstitution step. We have concentrated our work on how to obtain speedup results when this step is executed on parallel computing architectures.

On the surface the backsubstitution algorithm appears to be serial, but by exploiting the sparsity structure of the large sparse matrices, we have found that a speedup is attainable. The thesis shows, however, the speedup obtainable from the algorithm is significant.

The backsubstitution algorithm steps were represented by a data dependency graph. A node in the graph corresponds to an arithmetic operation, and arcs between nodes represent data transfers. We have develop a scheduling and simulation tool, called PARASIM, to aid in the required investigations.

PARASIM (Parallel Simulator) is a software program, which represents algorithms in data dependency graph, and simulates the operation of parallel computing architectures. It is an interactive environment in which one can simulate MIMD architectures of both distributed memory, as well as architectures with shared memory and interprocessor communication enhancement mechanism.

PARASIM has control structures for expressing parallel execution of algorithms, data transfer and models for hardware subsystems, such as processors, memory and bus structures. An intelligent scheduling method based on the critical path analysis concept was incorporated into PARASIM. This scheduling method ensures that minimum execution time is achieved. The key two factors which will affect the speedup are load balancing and interprocessor communication. It is essential that all processing nodes have equivalent computational loads, and the interprocessor communication is reduced as much as possible.

The thesis presents simulation results for a number of models exploiting different sources of parallelism. The thesis points out that communication enhancement hardware, such as one-to-all broadcast and caching, would improve the speedup by a factor of 10 to 20% respectively. sparsity structures and element interconnections.

ABBREVIATIONS

B U T P C	Bath University Transputer-based Parallel Computer
C A D	Computer Aided Design
M I S D	Multiple Instruction Single Data
M I M D	Multiple Instruction Multiple Data
C G	Conjugate Gradient
C P A	Critical Path Analysis
C P U	Central Processing Unit
F E M	Finite Element Method
F P U	Floating Point Unit
H L L	High Level Language
I C C G	Incomplete Choleski Conjugate Gradient
I P C	Inter-Processor Communication
L S I	Large Scale Integration
M I P S	Millions of Instructions Per Second
M M	Memory Module
O S	Operating Systems
PARASIM	Parallel Systems Simulator: A software program developed for this research by the author.
PDE	Partial Deffirential Equations
P E R T	Project Evaluation and Review Technique
P N	Processing Node
P P A	Parallel Processing Architecture
R A M	Random Access Memory
R I S C	Reduced Instruction Set Computer
S I M D	Single Instruction Multiple Data
S I S D	Single Instruction Single Data
T S B	Time-Shared Bus (multiprocessors)
T T L	Transistor Transistor Logic
V L S I	Very Scale Integration

List of Tables

Table of Figures

Table of Contents

Part ONE

Introduction and Background

This first part of the thesis is devoted to the presentation of introductory material on our research project, and to the discussion of some of the background to used throughout the other parts.

Part 1 comprises Chapters 1 and 2. Chapter 1 contains an introduction to the work carried out and identifies the different components used in this research. Chapter 2 contains some background information relating to the Conjugate Gradient algorithm.

Chapter 1

Introduction

1.0 Introduction

This thesis investigates the possibility of improving the performance of **CAD** systems by utilizing parallel computing architectures.

Introduction:

Computer Aided Design **CAD** software packages are available for engineers, architects or designers to facilitate their work. With the spread of **CAD** applications in both research and industry the demands set on these systems have grown. Requirements such as the capability to model complex design objects, sophisticated 3-dimensional colour graphics, user friendliness, fast response time or improvement of the basic functionality are always considered desirable. As a consequence the need for more computing power to perform these requirements also grows. Although the performance of uniprocessor computers is improving by utilizing more advanced RISC processors, there will come a time to utilize the power of parallel computing architectures to satisfy these needs. Conventional single processor computing systems are nearing fundamental speed limits, and will not be able to attain sufficient speedup. Parallel processing allows the application of a large number of processing nodes (more than one) to solve the same problem, and thus has the potential to achieve faster operation and response times.

CAD development cycle:

In some engineering applications delays can be tolerated in response time of the **CAD** system. In a research environment, however, it is often considered desirable to achieve fast solutions to the problems. Field solution for electromagnetic **CAD** systems involves the solution of a large set of linear

equations, and a typical design, simulation or analysis session involves repeated solution of the same model.

The linear system resulting from finite element method **FEM** has several characteristics that affect the choice of the solution method. One possible method for solving this system of equations is Incomplete Choleski Conjugate Gradient method **ICCG**. Parallel architectures offer extra computational power which can be utilized in enhancing the performance of **ICCG**. This thesis investigates the suitability of improving the performance of the **ICCG** algorithm on parallel computing architectures.

The need for rapid solution of large, sparse linear systems to be solved by the **ICCG** will be discussed in this thesis. But how is it possible to improve the execution of a serial program to be executed on a parallel computing architecture? To provide an answer for this question we need first to analyze the algorithm, namely **ICCG**, to identify potential components for parallelization, once these special components have been identified, then we will proceed to find the best method to attain the desired high performance.

Whereas, other research have looked into the problem from another point of view, that is the algorithm itself and how to make run efficiently on a parallel processing system, we investigated the scheduling of the algorithm to improve its performance.

We have analyzed the **ICCG** algorithm, and identified the portion of the algorithm which requires attention. This portion turned out to be the forward- and backsubstitution segment. Since the **ICCG** will iterate a number of times to solve the set of linear equations and during each iteration the backsubstitution will be executed once, then improving the speed of execution of this core segment will improve the overall performance dramatically.

We found out that the same sparse matrix structure will be used a number of times during a typical session. This finding coupled with the fact that the core segment is executed during each iteration of the algorithm has led us to focus our efforts to investigate further this matter. A closer look at matrix sparsity structures introduced patterns which may improve the parallel execution of the **ICCG**.

Our work in this research is concentrated on utilizing the matrix sparsity structures to identify the possible parallelism. In order to make the identification of parallelism possible, we have represented the different operations of the backsubstitution algorithm in a network structure.

In this network structure each operation, called *Node*, is a separate unit and it will receive its operands from other *Nodes* in the network depending on the data distribution and the position of the *Node* in the network in relation to matrix element distribution. The execution of the *Nodes* will depend on the availability of operands. Each *Node* can be executed only if the operands are ready and available. This network model has facilitated the presentation of the problem and enabled us to model the details of the operation and execution of the backsubstitution algorithm steps.

In order to obtain a measure for the available parallelism in the sparsity structure of the matrix, we have designed and written a software program that can measure such parameter. We have applied the Critical Path Analysis **CPA** strategy to identify the shortest sequence of *Nodes* that must be completed for the fastest execution possible of the network on parallel architectures.

Once the critical path within the network is identified the next problem will be to schedule all the nodes of the network to obtain the best speedup. The scheduling technique, which we have used in this research, is based on the following rule, which states that the *Nodes* residing on critical path are

assigned to one processor, whereas the other remaining *Nodes* are assigned to the remaining processors using a criteria which reduces the interprocessor communication time and produce the highest speedups. Implementing these scheduling rules will ensure the best possible results.

How to improve the performance of CAD systems:

In multiprocessor system performance evaluation, the system speedup and processor utilization are the most important aspects. In system selection or system comparison studies a certain level of quantification can be achieved by examining instruction rates and device speeds, but once again it is difficult to examine and compare how two systems will compare under the same algorithm. Simulating parallel computer systems will provide some quantitative answers to the performance of an algorithm on different architectures.

Thus, utilizing a simulation program to run the **ICCG** algorithm will produce quantitative results. These results may be used in comparing the suitability of parallel computing architectures.

1.1 Simulating Multiprocessors:

A detailed simulation that shows the performance, speedup and efficiency of different architectures was implemented in this research. Factors which could adversely impact the performance of the system in the full implementation are discussed, simulated and solutions suggested.

We have first to understand how different parallel architectures work, and propose a suitable simulation method to mimic their operation. Simulation models based on the use of Discrete Event Simulation DES are best suited for this class of digital systems. Parallel computing architectures are digital systems that state transitions take place during the clock change. These digital systems are best simulated using the DES models. Creating different models to describe the behavior of the computer subsystem and combining them in one system will result in a model for parallel architectures.

The target machine, we simulated for this research, is assumed to have a general **MIMD** architecture composed of a number of Processing Nodes or elements PN. Beside containing local memory to hold code and data, each PN incorporates a private communication mechanism. The program models: Processing node, local and remote memory, communication architecture, bus operation and others. We have simulated both the distributed and the shared memory system architectures.

Distributed Memory Architecture:

The distributed memory system model is assumed to be a fully interconnected structure, where each processing node can communicate with any other processing node by direct link.

Shared Memory Architecture:

The architecture, we intend to simulate, features a number of processing nodes connected by a multiprocessor time-shared bus. Each processing node is partitioned into an arithmetic unit, which performs the actual computations, and a communications unit, to provide the necessary hardware to interchange of data between the processing nodes.

Special features:

We have added special hardware enhancement features into our simulator to simulate a bus with the characteristics:

1- High bandwidth, to alleviate the communication bottlenecks as far as possible.

2- The ability to support communication enhancement features to improve Interprocessor Communications IPC. These two functions will be utilized by the shared memory bus system are one-to-all Broadcast and caching architectures.

The simulator will be used to investigate the effect of the sparsity structure on the performance of parallel execution of the ICCG algorithm. This simulator, which will be equipped with an option to allow intelligent scheduling of tasks, will be used to investigate the effect of the sparsity structure on the performance of parallel execution of the ICCG algorithm.

Research Objectives

The objectives of this research project are as follows:

1- Identify the computational bottleneck of the **ICCG** algorithm.

2- To measure the performance of **ICCG** algorithm on parallel architectures using simulation.

3- This requires the simulation of different types of parallel computer architectures.

4- Write a simulation program which enables the user to simulate the algorithm running on different architectures.

5- Apply communication enhancement modules and measure their performance effects.

1.2 Achievements

1- We have applied the **CPA** strategy to the scheduling of a numerical algorithm to improve its performance on a multiprocessor system.

2- Obtained good results from the simulation that show it is possible to benefit from the intelligent scheduling method.

3- We have experimented with different data interconnections.

4- The granularity of the network played an important role in its performance, thus, we have experimented with different forms and granularity structures. We have found by simulating the algorithm on different granularity sizes, that the performance was enhanced by grouping Nodes together.

5- The performance of the serial implementation of the scheduler and simulator was improved by a number of programming techniques to minimize the execution time.

1.3 Thesis overview

first shows the different components that were used to develop and execute this research. The components are related to: The numerical algorithm, Scheduling, Critical path analysis, and Multiprocessor architectures. These components are discussed in more detail in this thesis which is divided into five main parts.

Part One: Introduction and Background.

Chapter 1 introduces the work carried out and our objectives.

Chapter 2 depicts the Incomplete Choleski preconditioned Conjugate Gradient (**ICCG**) algorithm, which is the basis for the solution of the large sets of linear equations encountered in numerical field modeling.

Part Two: Sturcture and Simulation of Parallel Computing Architectures

Chapter 3 introduces the diverse hardware architectures for parallel computers and gives examples of each category.

Chapter 4 surveys previous attempts of simulating uni- and multiprocessor architectures. A comparison table is prepared and showed to summarize past research.

Part Three: Scheduling and Simulating

Chapter 5 describes the Graph theory and how it is used to allocate parallel programs.

Chapter 6 presents the structure and internal operation of the scheduling and simulation program PARASIM. Forms of user-interface, program output, presentation of algorithms and different parameters of simulation are also discussed.

Part Four: Results and Conclussions

Chapter 7 presents and discusses results for the simulation of the Incomplete Choleski Conjugate Gradient algorithm ICCG on the software simulator.

Chapter 8 concludes the work carried out to investigate the simulated parallel implementation of ICCG algorithm, and gives suggestions for the possible future work and system improvements.

Part Five: Appendices

Appendix-**A**: Speedup results tables for different configurations and architectures.

Appendix-**B**: List of main routines of the PARASIM simulation program.

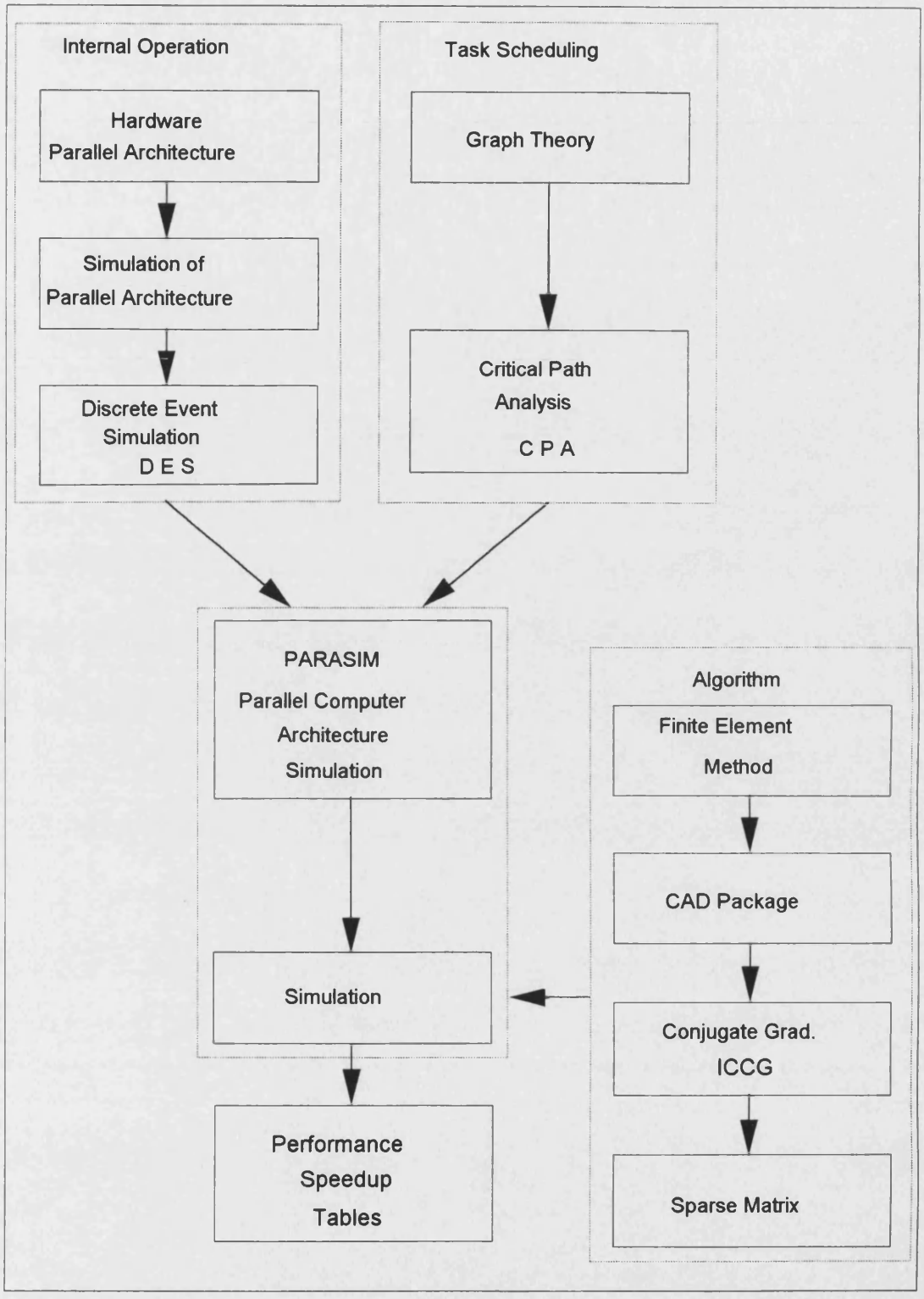


Figure 1: Organization of this thesis.

Chapter 2

Conjugate Gradient Algorithm

2.0 Conjugate Gradient Algorithm

This chapter introduces the Conjugate Gradient numerical algorithm used in the solution of the Field equations. This thesis investigates the possible improvements in execution using parallel architectures for the algorithm.

2.1 Background information

Finite Element Method (FEM) calculations have a number of characteristics which make them a candidate for distributed processing systems:

a) Each run involves a lot of arithmetic.

b) Each run involves a data storage.

c) Typical problems involve a number of runs (to check the effects of a variety of loadings, transient and nonlinear).

d) There is a desire to use ever more complex runs (e.g., 3-D calculations).

Let us now identify the typical FEM calculation steps:

1- *Discretisation:* the region is subdivided into a number of elements.

2- *Linearization:* a scheme is chosen for handling the nonlinearities (usually, linearization and iteration of some sort).

3- *Approximation:* the Partial Deffirential Equations PDE system is replaced by an approximate finite set of equations involving a number of unknowns; each of these is associated with one or a few elements.

4- *Partitioning:* the unknowns are partitioned amongst the available processors.

5- *Assembly:* the matrix corresponding to the chosen Discretisation is assembled.

6- *Solution:* the unknowns are computed as the solution of these equations. This step is the most time consuming. Rapid results are very important to the computer user.

Stages 5 and 6 may be repeated as the iterations for nonlinear and/or time transient problem proceed. The solution phase is usually the most time consuming section and requires computational power to be completed in shorter times.

It would be helpful if we could identify which portion of a typical FEM calculation will be enhanced by the utilization of a parallel architecture computer. Thus, investigating the solution step to find the portion which consumes a lot of computation time, will lead to great improvement in the overall performance of the system.

The equation is discretized over each element, resulting in a set of linear equations for the values of the variables at the nodes contained on that element. These equations are then combined for the entire problem, resulting in a single large linear system of the form

$$A.x = b \quad \text{... Equ. 1}$$

Where A is a matrix describing the structure of the system and the characteristics of each element, x is a vector of the values of the variable of interest at all nodes, and b is a vector of the boundary conditions applied to the system.

The Equation Solver:

Consider the task of iteratively solving *M* simultaneous equations. We may regard this problem as equivalent to finding the M-dimensional vector which minimizes some residual error quantity defined on the M-dimensional

space. A gradient method makes use of trail values for the variable at step i to generate new values at step *i+1* corresponding to reduced value of the error function.

By successively moving "downhill" toward the error minimum, the method converges toward the desired solution. The *conjugate gradient* method Jennings [Jennings-77] employs descent direction vectors that are mutually orthogonal.

Property of *A* matrix

a- Sparse:

Once the *A* matrix has been constructed, a typical model with 10,000 nodes each row will have a large number of non-zero entries, so more than 99.00% of the entries in the A matrix will be zeros. This sparsity can and must be exploited in order to obtain an efficient solution.

b- Symmetric:

A significant property of the linear system comes from the fact that the model represents a physical system. In this case, the constant relating the field value at node *b* to that at node *a* is the same as the constant relating the values in the reverse order. If $a_{i,j}$ represents the value of the matrix entry for row i and column j, then $a_{i,j}$ will be the same as $a_{j,i}$ in other words the matrix is symmetric. Also, in most applications, the finite element method gives rise to linear systems which are positive definite (matrix A is said to be positive definite if, for any vector *x* except the zero vector, $x^T A x > 0$.

c- Positive:

In most applications, the finite element method gives rise to linear systems which are positive definite (matrix *A* is said to be positive definite if, for any vector x except the zero vector, $x^T A x > 0$. Matrices which are

symmetric and positive definite have important properties for solution, which will be described below; they are referred to as SPD matrices.

d- Large:

The linear systems covered by this research are large. The range currently available computers can solve systems of 1,000 to 100,000 or more nodes, depending on the time for solution considered tolerable.

2.2 The Conjugate Gradient Algorithm

The **Conjugate Gradient CG** method has proven to be the appropriate for the solution of the linear system of equations arising from the **Finite Element Method** (FEM).

The **CG** method, used in solving FEM models, was developed by Hestenes and Stifel [Hestenes-52], but did not achieve wide use in engineering analysis until the 1970's. In the last decade, the method has gained considerable popularity and has been used for a variety of applications.

The **CG** method insures that the correct solution vector x will be found in at most N_{eq}, excluding round-off error and using exact arithmetic. Reid, however, found that round-off error can significantly deter this convergence, especially for system with a high condition number (10^4 - 10^5). In these cases, the number of iterations may greatly exceed N_{eq}, with rapid convergence occurring only near the end of the iteration process. In some cases due to computer number presentation and accuracy the covergence will not take place.

The **CG** is an iterative method, i.e. one which is repeated to generate successively better approximation solutions to a problem. In the method of conjugate gradients the direction vectors are chosen to be a set of vectors

$p^{(0)}$, $p^{(1)}$, etc., which represent, as nearly as possible, the directions of steepest descent of points $x^{(0)}$, $x^{(1)}$, etc., respectively, but with the overriding condition that they be mutually conjugate.

2.2.1 The C G iteration

The **CG** method without preconditioning steps are described as follows. To solve $A.x=b$, where A is a positive definite symmetric ($n \times n$) matrix:

1- Set an initial approximation vector $x^{(0)}$,

2- Calculate the initial residual $r^{(0)}=b-Ax^{(0)}$ ... Equ. 2

3- Set the initial search direction $p^{(0)}=r^{(0)}$ ... Equ. 3

4- then for *i=0, 1, ...*

4.1- Calculate the coefficient

$$a_i=\frac{[p^{(i)}]^T r^{(i)}}{[p^{(i)}]^T A p^{(i)}} \text{ ... Equ. 4}$$

4.2- Set the new estimate

$$x^{(i+1)}=x^{(i)}+\alpha_i p^{(i)} \text{ ... Equ. 5}$$

4.3- Evaluate the new residual

$$r^{(i+1)}=r^{(i)}-\alpha_i A p^{(i)} \text{ ... Equ. 6}$$

4.4- Calculate the coefficient

$$\beta_i=\frac{-r^{(i+1)} A p^{(i)}}{[p^{(i)}]^T A p^{(i)}} \text{ ... Equ. 7}$$

4.5- Determine the new direction $p^{(i+1)}=r^{(i+1)}+\beta_i p^{(i)}$, continue until either $r^{(i)}$ or $p^{(i)}$ is zero.

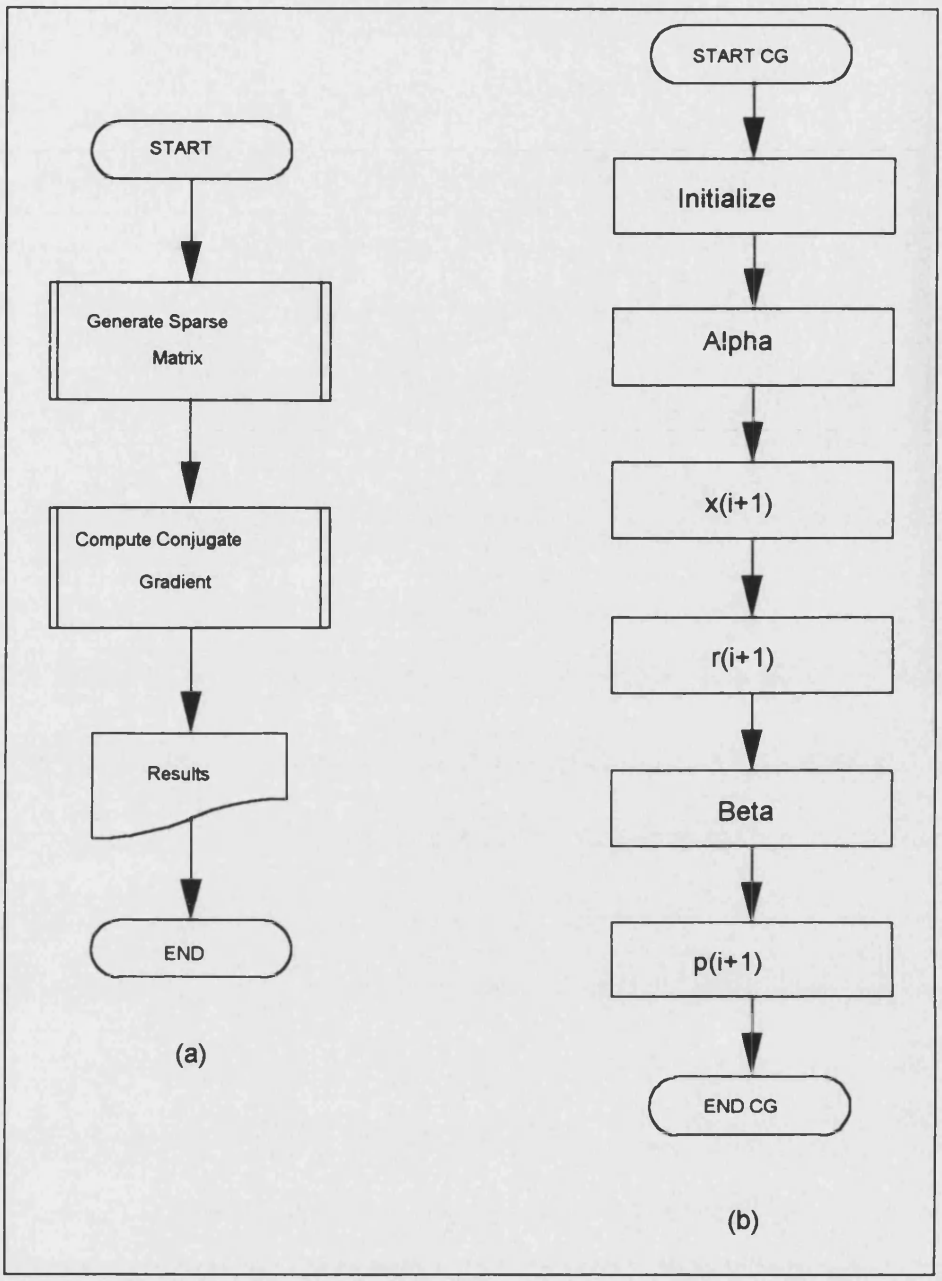


Figure 2: Flowcharts for a) CG main program, b) One CG iteration.

The Conjugate Gradient CG algorithm uses the following vectors which need to be updated at each iteration:

$p^{(n)}$ conjugate direction vector,

$r^{(n)}$ residual vector,

$x^{(n)}$ solution vector,

$u^{(n)}$ product of transformed matrix and conjugate direction vector.

The flowchart in FIGURE 2 A shows the steps needed to setup and execute a typical CG action, and FIGURE 2 B shows the detail steps or the execution of one CG iteration. The vector $r^{(n)}$ is the residual vector of the system. The search vector $p^{(n)}$ is the direction from the current estimate of the solution along which one moves a distance to obtain a new solution estimate.

2.3 Parallel Features of C G

In this section we will discuss the features of the Conjugate Gradient algorithm that enables a distributed implementation.

Parallel CG introduction:

The **CG** is well suited for parallel processing. Each step can be executed in parallel, but the steps themselves must be done in order. Therefore, the computational load may be easily balanced between the processing nodes. The calculations are comprised mainly of the matrix-vector multiplication, vector additions and subtractions and vector dot products. We use the conjugate gradient algorithm to solve a sparse system A x = b, where *A* is an *n X n* matrix with *m* nonzero entries ($n \leq m << n^2$).

Parallel CG implementation:

These operations are easily implemented in parallel with a minimum of communication required between processors. For example, each vector of length may be divided amongst all the processors, with each processor then performing the vector operation on its portion of the vector. Communication of scalar quantities is required during the dot products and the convergence test. No communication is required for the vector additions and subtractions.

Dot product analysis:

Each processor needs additions to complete the dot product on its local vectors. These partial results are added up globally, and the result is made available to all the processors. This operation can be parallelized only in part. A broadcast mechanism will improve highly this operation.

Matrix by vector multiplication:

The *m* multiplications and the *m-n* additions in the sparse matrix by vector multiplication can be perfectly parallelized if *A* is partitioned row-wise and if the rows are distributed in a way which balances equally the nonzero entries. This is affected by the average number of elements in the row. Every processor gets all the rows corresponding to the part of the vectors it owns, thus producing again the same part of the resulting vector.

Parallel Features of CG:

Equation in step **4.1**: uses the special purpose parallel matrix-by-vector multiplier. The inner products in the numerator and denominator are calculated in parallel.

Equation in step **4.2**: updates all the elements of x simultaneously using termwise operations. The scalar α_i is mapped onto a vector with each entry equal to α_i, and then multiplied with p_i using termwise multiplication. The resulting vector is then added to the vector x_i in parallel using termwise addition.

At each iteration, the basic algorithm involves the following mathematical operations:

a- One multiplication of the sparse matrix *A* by a vector,

b- Two vector dot products,

c- Three multiplications of a vector by a scalar,

d- Three additions of vectors,

e- Two scalar divisions,

f- Test for convergence.

The divisions and some flow control (including the convergence test) form the serial part of the algorithm. Linear operations on vectors and scalar multiplications inside the vector dot product can be perfectly parallelized on p processors when each processor owns $\frac{n}{p}$ elements of each vector.

Data Types:

Three types of information which must be stored are categorized as follows:

1- *Definitely Global* Data which must be available to all processing nodes in the system, such as α and β.

2- *Preferably Global*: Data which would most conveniently be available to all processing nodes. The contents of the vectors used by **CG** except those which are definitely global fall into this category. If this information is not globally available then either it must be stored a number of times elsewhere or individual processors must be assigned specific tasks before execution commences.

3- *Local*: Information such as program code, intermediate results of operations, and stack contents, which need only be available to individual processors.

For a specific hardware implementation, the choice between storing preferably global data in locally accessible or globally accessible memories will be the result of consideration as to whether or not bus contention problems involved when using globally accessible memory outweigh the problems of local storage.

2.5 Preconditioned Incomplete Choleski's Conjugate Gradient ICCG

The Conjugate Gradient algorithm performs well for a wide variety of applications. However, a large condition number severely retards the convergence rate of the algorithm, In addition, if the eigenvalue spectrum of A has values that are evenly distributed, rather than being clumped near each other, the Conjugate Gradient algorithm tends to converge more slowly. To improve the condition of the linear system of equations, preconditioning may be used. The Choleski decomposition method produces a matrix with all elements filled. This destroys the sparsity structure of the original matrix. Some off-diagonal elements are inspected to determine whether any need to be deleted according to whatever deletion criterion is to be used, This leads to the Incomplete Choleski decomposition. The ICCG algorithm becomes

$$u^{(k)} = L^{-1} A L^{-T} p^{(k)} \quad \text{... Equ. 8}$$

L is an approximation inverse, which can be constructed from the Choleski decomposition. However only the terms which correspond to a non zero entry in the A are kept. L is thus an approximate inverse of A.

$$\alpha_k = \frac{[r^{(k)}]^T \cdot r^{(k)}}{[p^{(k)}]^T \cdot u^{(k)}} \quad \text{... Equ. 9}$$

$$y^{(k+1)} = y^{(k)} + \alpha_k \cdot p^{(k)} \quad \text{... Equ. 10}$$

$$r^{(k+1)} = r^{(k)} - \alpha_k \cdot u^{(k)} \quad \text{... Equ. 11}$$

$$\beta_k = \frac{[r^{(k+1)}]^T \cdot r^{(k+1)}}{[r^{(k)}]^T \cdot r^{(k)}} \quad \text{... Equ. 12}$$

$$p^{(k+1)} = r^{(k+1)} + \beta_k \; p^{(k)} \quad \text{... Equ. 13}$$

The pre-multiplication step is obtained without forming the transformed matrix explicitly by the following three operations:

Back substitution $L^T v^{(k)} = p^{(k)}$ ... Equ. 14

Pre-multiplication $w^{(k)} = A v^{(k)}$ ... Equ. 15

Forward substitution $L u^{(k)} = w^{(k)}$ ... Equ. 16

Most of the above steps of ICCG can be executed utilizing parallel and vector processing architectures, but the last 3 steps would not benefit from such architectures. This is due to the irregular and sparse structure of the matrix involved. The sparsity structure of the used matrix introduces elements with zero value, thus a method which will not take into account such sparsity structures will not produce good and efficient results.

A critical segment of the ICCG algorithm is the solution of the Forward and Backsubstitution using the L and L^T triangular matrices. Since we are going to keep the same sparsity structure of the original matrix ...

2.6 Discussion

Let us now clarify the principles and observations discussed so far. Our intention is to investigate the suitability of parallel architectures for the execution of ICCG. We have identified the backsubstitution segment in which we will improve its execution on parallel architectures. We will consider two simple examples of the detailed execution of the Backsubstitution algorithm.

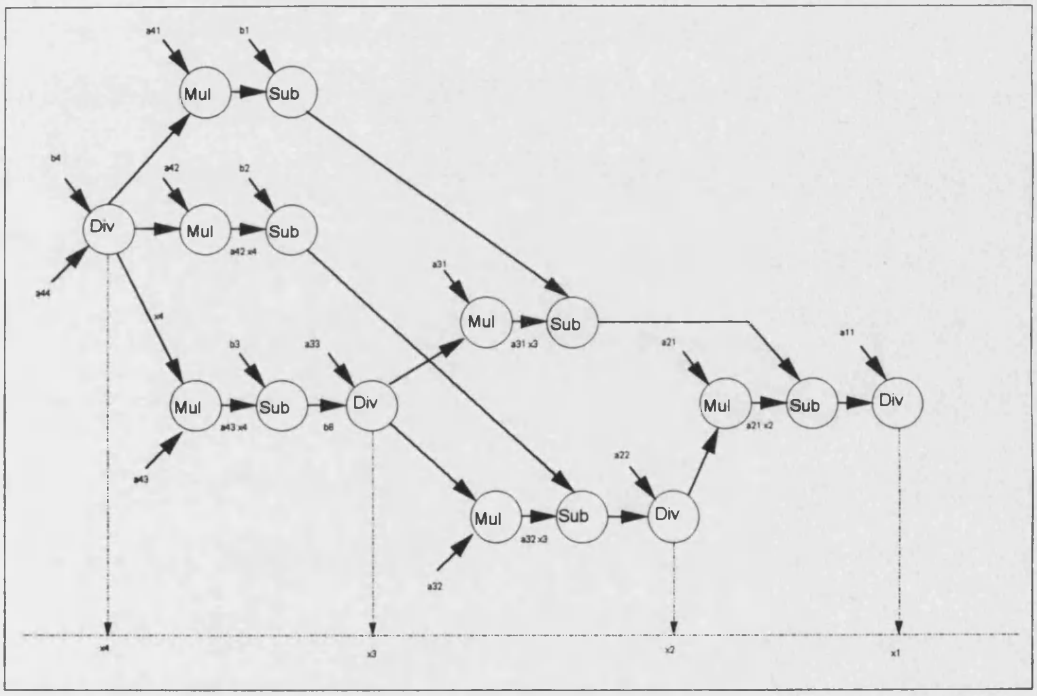


Figure 3: Dataflow details for a full 3x3 matrix.

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ & a_{2,2} & a_{2,3} & a_{2,4} \\ & & a_{3,3} & a_{3,4} \\ & & & a_{4,4} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \quad \text{... Equ. 17}$$

The first example is depicted by the above equation. Utilizing the Backsubstitution algorithm we can get the values of x. When the steps to solve the equation are organized in a network, where each circle represents an arithmetic operation and the arrows represent the data and operand movement, then we get FIGURE 3. In this figure the values of x array are produced in the following sequence: x_4, x_3, x_2, and at last x_1. This sequence of generating the x values is dictated by the matrix structure. In this example the matrix is fully populated and the solution is inherently serial.

$$\begin{pmatrix} N^{24} & 0 & 0 & 0 & 0 & 0 & 0 & N^{12} & 0 & N^{2} \\ & C^{23} & 0 & C^{21} & 0 & 0 & 0 & 0 & 0 & N^{3} \\ & & N^{22} & 0 & N^{19} & 0 & 0 & 0 & 0 & N^{4} \\ & & & C^{20} & 0 & C^{17} & 0 & C^{13} & 0 & N^{5} \\ & & & & N^{18} & 0 & N^{15} & 0 & 0 & N^{6} \\ & & & & & N^{16} & 0 & 0 & N^{8} & 0 \\ & & & & & & N^{14} & 0 & N^{9} & 0 \\ & & & & & & & C^{11} & C^{10} & 0 \\ & & & & & & & & C^{7} & 0 \\ & & & & & & & & & N^{1} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \\ b_8 \\ b_9 \\ b_{10} \end{pmatrix}$$

Let us now consider another example with a sparse matrix structure. The above equation depicts the second example equation. Figure 4 shows the network relationships and dataflow derived from the above equation.

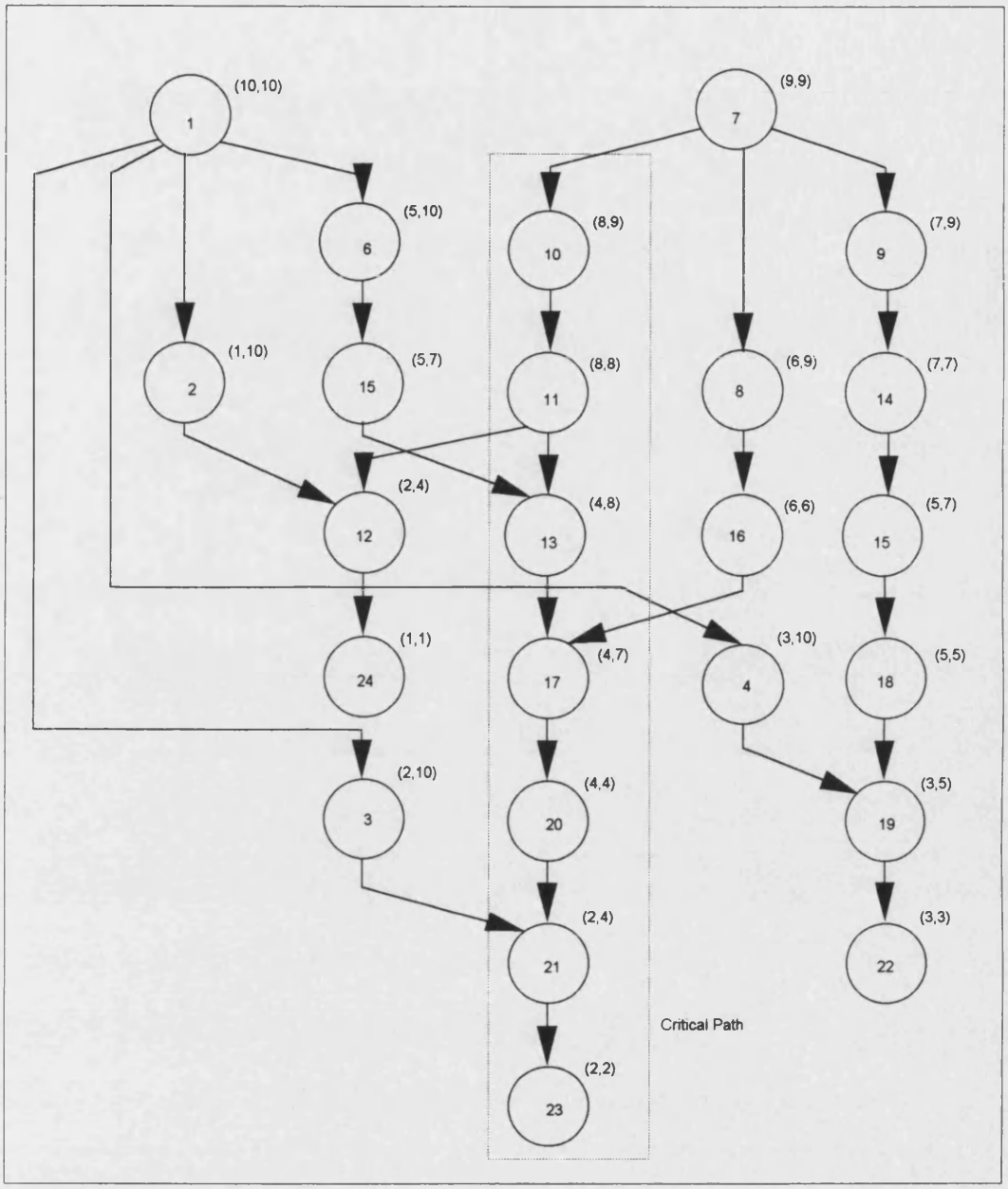


Figure 4: The dataflow for the example sparse matrix size 10x10.

The matrix contains 24 element. Each element of the matrix is marked with a unique number starting from 1 to 24. If the above equations is to be executed on a parallel computer, then what is the minimum time needed to complete it execution? In order to find this time we have to find the sequence of elements that are dependent on each other. The sequence of nodes marked with the letter *C* is the sequence that must be followed during

execution for the least time. The sequence that follow the elements: 7- 10- 11- 13- 17- 20- 21 and 23 is known to be the critical path, which starts at element 7 and ends at element 23. The execution can not be faster than the identified sequence. More on this topic will follow in the thesis.

2.7 Conclusion

The CG algorithm for the solution of large sparse linear equations has been presented in this chapter. Both the serial and parallel code was discussed. The CG algorithm lends itself easily to parallel implementation and will utilize vector processing hardware efficiently. Improving the results from the CG has led to the need for factorization. The ICCG introduced extra matrix operation steps.

The parallel implementation of the ICCG algorithm is not straight forward. It needs attention is both the forward and backsubstitution segments. This is because of the sparse matrix structure involved. We have to utilize the sparsity structure of the matrix to obtain parallelism within these segments.

The structure of the matrix used is both sparse and irregular, this will not enable the application of Sandra parallel programming techniques.

In order to investigate the behavior of ICCG algorithm on parallel architectures, we have to device a simulation method which will mimic the operations of a multiprocessor system architecture and allows the algorithm to be simulated. The purpose of the simulator is to give quantitative measurements for the performance of the algorithm. Adding communication features to the simulator will provide a tool to experiment with different architectures.

2.8 References

[Ajiz-84] Ajiz, M. A. and Alan Jennings, "A robust incomplete Choleski-Conjugate Gradient Algorithm", *International Journal for Numerical methods in Engineering*, 1984, Volume 20, pp. 949-966.

[Hestenes-52] Hestenes, M. and E. Stiefel, "Methods of the Conjugate Gradients for solving linear systems", *Journal of research of the national bereau of standards*, Volume 49, Number 6, December 1952, research paper 2379, pp. 409-436.

[Jennings-77] Jennings, Alan, "Matrix Computations for engineers and scientists section 6.13", John Wiley & Sons, ISBN 0 471 99421 9, 1977, pp. 212-222.

[Kershaw-78] Kershaw, David S., "The Incomplete Choleski - Conjugate Gradient Method for the Iterative Solution of Systems of Linear Equations", *Journal of Computational Physics*, vol. 26, pp. 43-65, 1978.

[Magnin-89] Magnin, H. and J. L. Coulomb, "A Parallel and vectorial implementation of basic linear algebra subroutines in iterative solving of large sparse linear systems of equations", *IEEE transactions on Magnetics*, Vol. 25, No. 4, July 1989.

[Munksgaard-80] Munksgaard, N., "Solving sparse symmetric sets of linear equations by preconditioned conjugate gradients", *ACM transactions on mathematical software*, vol. 6, no 2, June 1980, pp. 206-219.

[Poon-89] Poon, K. M., "Theoretical study of parallel processing for the solution of matrices using Conjugate Gradient", School of Electrical engineering, B. Sc. final year project, University of Bath, 1989.

[Strang-86] Strang, Gilbert, "Introduction to applied mathematics", Wellesley - Cambridge Press, ISBN 0-961-4088-0-4, 1986, pp. 378-379, 418-425.

Part TWO

Structure and Simulation of Parallel Computing Architectures

The ICCG algorithm will run on a parallel architecture system, thus we need to investigate the different forms and styles available such systems, and understand the problems associated with distributed programming. In this part of the thesis the different structures of the parallel computing architectures will be demonstrated.

Chapter 4 will be devoted to survey previous work in the field computer systems architectures simulation..

Chapter 3

Parallel Computer Architectures

3.0 Parallel Computer Architectures

In this chapter we will introduce the concept of *speedup,* different forms of parallel architecture systems and their classifications. Factors that prevent the realization of linear speedup are also discussed. A closer look at the dataflow architecture with examples is also illustrated.

Introduction

The advent of VLSI technology and low-cost microprocessors has made distributed computing an economic reality in today's computing environments. The modularity, flexibility and reliability of distributed processing makes it attractive to many types of users. Several distributed processing systems, which posses high throughput, have been designed, implemented and sold during the past decade.

The motivation for using parallel processors in scientific applications is to provide a speed improvement over single processor systems. An improved system response and powerful processing capabilities would improve research in such applications.

Applications:

Distributed processing applications range from data base installations, where processing load is distributed for organizational efficiency, to high-speed signal processing systems, where extremely fast processing must be performed under real-time constraints. Distributed processor systems are currently used for advanced, high-speed computation in application areas such as Computer Aided Design (CAD), image processing, artificial intelligence, signal processing and general data processing.

Other areas of application of parallel algorithms include matrix computations, sorting and searching, FFT, partial differential equations and optimization.

The problem of Degradation:

The use of distributed and parallel processor computing systems today requires system designers to partition an application into at least as many functions as there are processors. Spare processors must be allocated to useful jobs within the system. But, like any other concept, distributed processing has problems which must be solved in order to benefit from its advantages.

This brings up the issue of performance measurement. How should we characterize the performance of a parallel computer when in effect, parallel computing re-defines traditional measures such as *MIPS* (Million Instructions Per Second) and *MFLOPS* (Million Floating Point Operations Per Second)? A new measure of performance is needed to relate parallel computing to performance.

The Speedup curve

The most often quoted measure of parallel performance is the *speedup curve.* This is computed by dividing the time to compute a solution to a certain problem using one processor, by the solution time using N processors in parallel.

$$speedup = \frac{\text{time required for the nonparallel program execution}}{\text{time required for parallel version execution}} \quad \textbf{... Equ. 18}$$

The maximum speedup possible is always the number of processing nodes used to implement the problem. However, in practical implementations

it is often less than this. The speedup as a measure for the performance of parallel programs execution, describes how efficiently the multiprocessor system is utilized. for instance, if a parallel program has a speedup of 8 and is making use of eight processing nodes, then the parallel program is as efficient as possible. On the other hand, if the attained speedup is only 2.56 for 8 processing nodes, then it is making less utilization of the available processing throughput. This introduces a degradation of the system performance.

A serious problem in distributed processing is the degradation in throughput caused by the system overheads and interprocessor communications (IPC). In an ideal multiprocessor, we would expect throughput to increase linearly as the number of processors increases. That is, we expect to double throughput by doubling the number of processors available to perform a task, if the modules being processed are independent of each other (excluding the effects of control overheads). This expected performance is shown by the ide*al curve* in FIGURE 5.

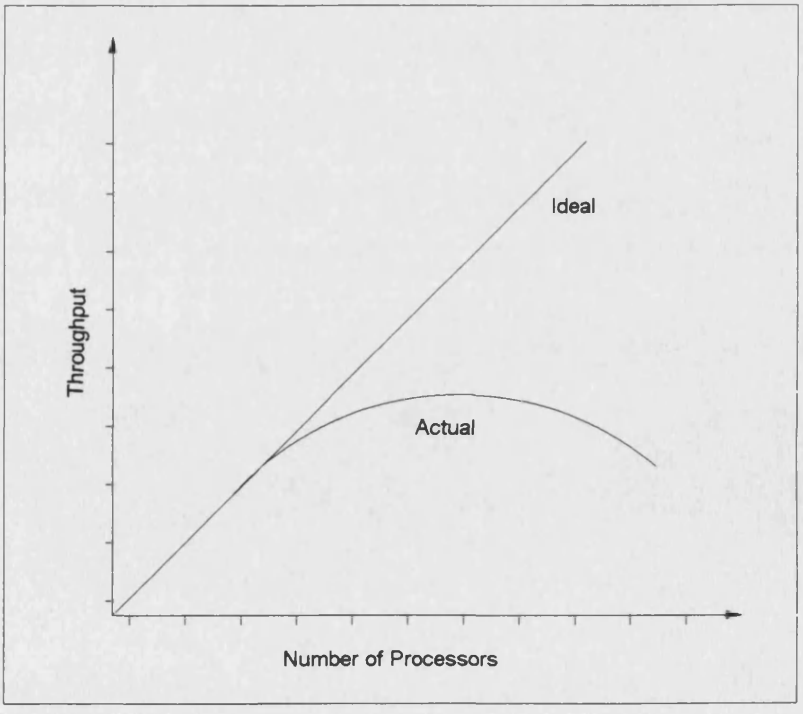


Figure 5 Throughput of a multiprocessor computing system.

In practice it is difficult to attain this ideal speedup owing to the following factors:

1- Inherent characteristics of the solution algorithm which limit parallelism available in a particular problem solution.
2- Overhead associated with exchanging or sharing data among the processors, together with,
3- Overhead required to control the parallel processors, operating system overheads, and the finite time needed for the transfer of data among the processing nodes.

A discussion about the execution of algorithms on parallel processing systems must begin with a review of existing parallel processing architectures. The following section will give a brief review which will identify and examine the different parallel processing architectures available.

3.1 Multiprocessor Computer Systems

The parallel computer architectures may be divided into four broad categories, given their method of handling instruction streams per cycle and their method of handling data. The four categories are:

- Single-Instruction Single Data (**SISD**)
- Single-Instruction Multiple Data (**SIMD**)
- Multiple-Instruction Single Data (**MISD**)
- Multiple-Instruction Multiple Data (**MIMD**)

SISD machines are sequential processors, termed Von Neuman machines and are common in the computer field. Processors ranging from Personal Computers (PC) through super-mini computers to mainframes all possess SISD architecture (e.g., IBM PC-AT, VAX 6000, VAX 8700, IBM 4341). Some computer systems incorporate pipelining within their arithmetic units. This concept means that the system breaks down an operation such as

multiply into a number of steps, each of which can proceed concurrently on distinct operands. If independent arithmetic units exist, several operations can proceed simultaneously. Breaking arithmetic operations into sections requires some additional hardware and results in its own increased overheads. We will show later in this thesis how the pipelining concept was introduced to our simulator.

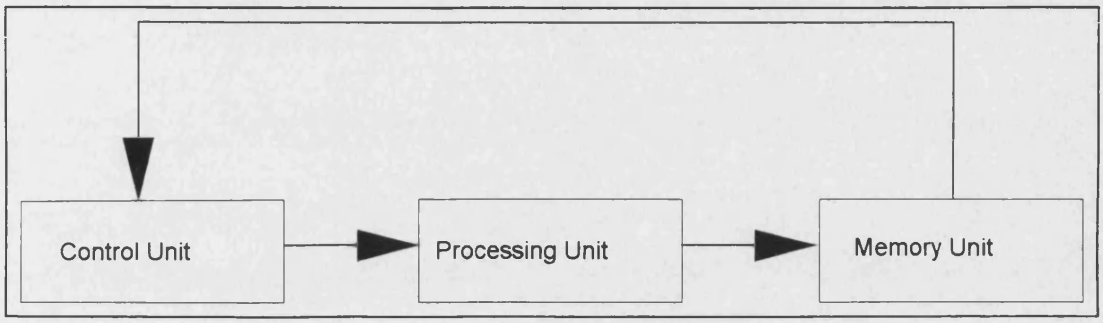


Figure 6: SISD computer architecture.

SIMD machines include computers which can perform one operation (instruction) simultaneously on multiple data. For example, the vector supercomputers established in the 1970s (e.g., Cray-1) are SIMD machines. Multiprocessor systems which enable each processor to execute the same instruction simultaneously, but each with a different set of data, are termed SIMD environments (e.g., the Connection Machine). Pipelined array processor architectures (e.g., Floating Point Systems 264 array processor) may be grouped into this category as well, although the amount of data operated upon in a single instruction is much smaller than on the vector or multiprocessor computers.

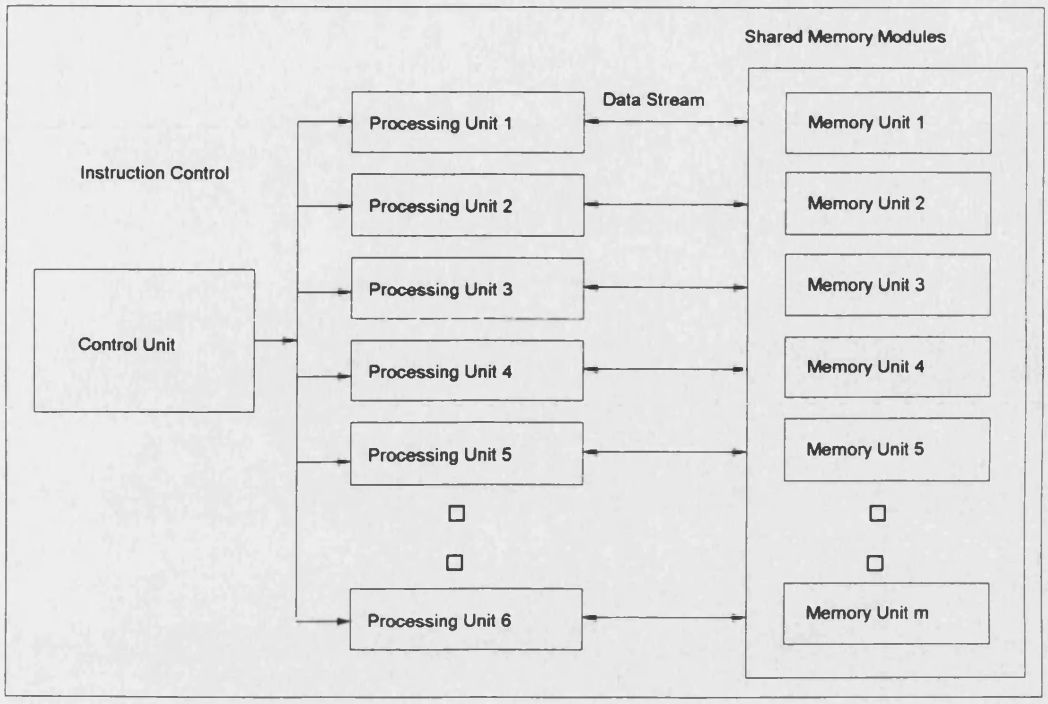


Figure 7: SIMD computer architecture.

MISD machines involve a chain of processors and are similar in design to SISD systems, although pipelined processors are sometimes considered to be in this category. Finally, **MIMD** architectures may execute multiple instruction streams on multiple data. Technically, any multiprocessor system which does not require each processor to execute the same instruction simultaneously may considered to be of MIMD architecture. The processing nodes (PNs) of these multiprocessor systems must be connected in some configuration either directly by communication links or through the sharing of global memory.

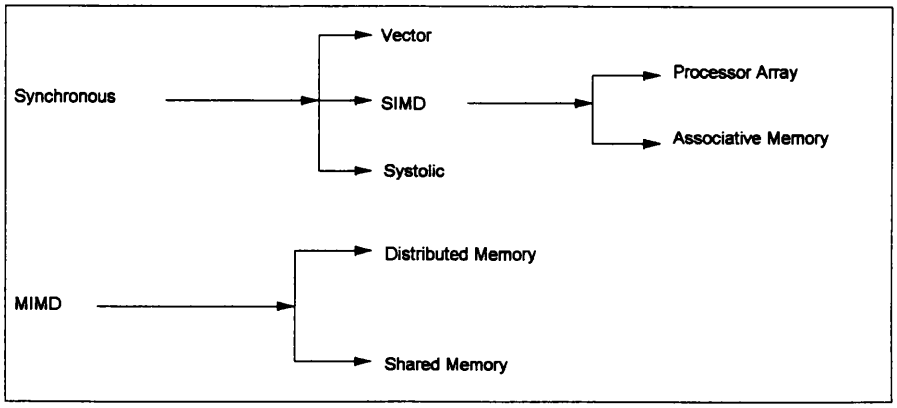


Figure 8: High-level taxonomy of parallel computing architectures.

MIMD multiprocessor systems can be partitioned as follows:

1- Shared Memory Multiprocessors.

2- Distributed Memory Multiprocessors.

3- Dataflow architecture.

3.1.1 Shared Memory Multiprocessors

One method of communication among a number of processors is by shared memory. In this category all the processing nodes PNs are connected to a common memory system either by a direct connection through a network of connections or via a memory bus which is essentially a channel along which processors pass requests for data and the memory unit returns the requested data back to the processing units. In this system every data location is available to every processing node, if a processing node needs data for its task it simply reads it from the memory. As other processing nodes may be continually modifying the stored values of the data, care must be taken that no processing node accesses the memory location before an appropriate value has been stored. A common method is to reserve certain areas of memory to keep track of whether locations have yet been written to in order to safeguard against premature attempts to gain access to data these locations do not yet contain.

Limitations of Shared Memory Systems

The major limitation with shared memory systems is the difficulty and expense of allowing a large number of processing nodes to concurrently access the memory system. Direct connections of large number of processors is physically extremely difficult to arrange, but on the other hand a common bus system has to be very fast to service memory requests from all the processing nodes at once.

3.1.2 Mechanisms to improve system performance

There are a number of hardware/software mechanisms that improve the performance of shared memory multiprocessor systems. These mechanisms may improve the computation and/or communication capabilities of the system. Among these mechanisms that improve interprocessor communication are the following:

1- One-to-All broadcast operation.

2- Caching logic.

One-to-All Broadcast

One method of improving the performance of shared memory multiprocessor systems is One-to-All broadcasting. In this method of communication a special bus cycle in implemented. In this cycle data put on the bus by the sending processing node PN will be stored in all processing nodes residing on the bus. Broadcasting from one node to all other nodes ensures that when the data is needed, the PN will not fetch it from another remote PN but will find it in its local store.

Using Cache mechanisms

In order to improve the shared memory system performance the concept of cache architectures is introduced. A schematic illustration of the place of cache in the processing node hardware is in ninth. In this mechanism

the cache controller will ensure that location specified by the software will continuously be updated. This is made by observing the bus for specific addresses.

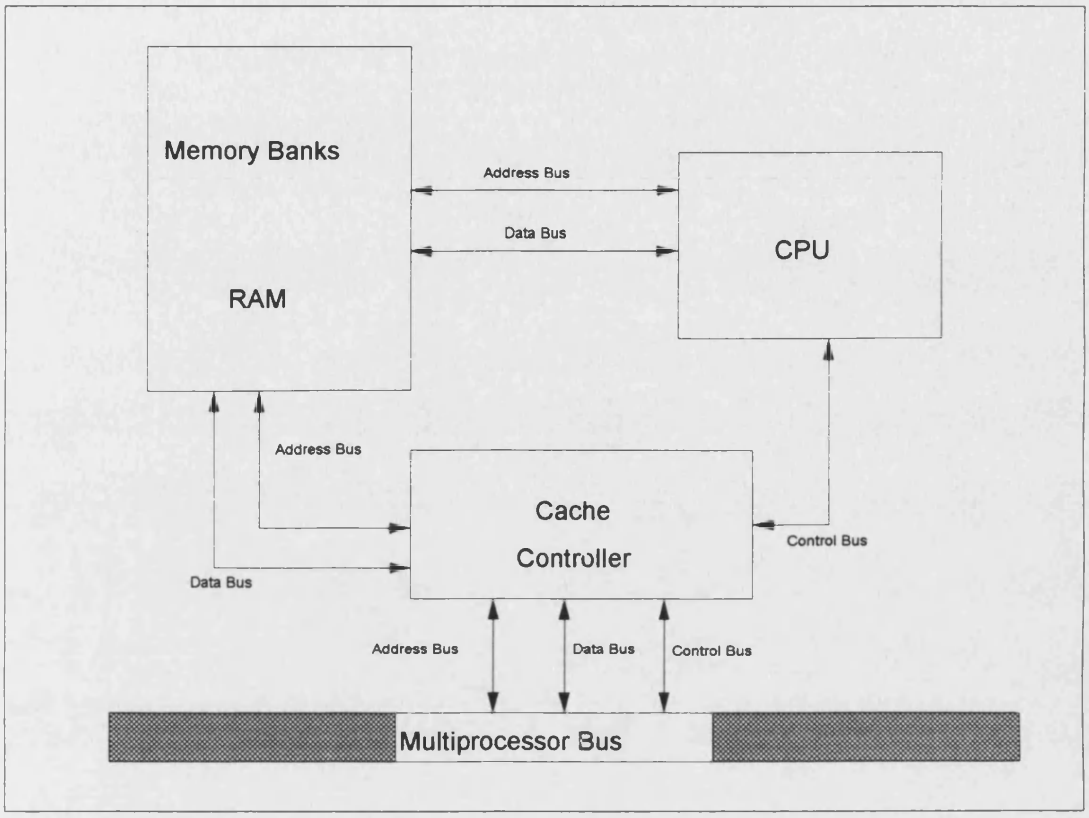


Figure 9: Cache architecture for the Processing Node.

3.1.3 Distributed Memory Multiprocessors

A completely different approach to communication is that of distributed memory architectures. In these each processing node has some amount of memory attached to it and the processor-memory pair are connected in one of a number of schemes. Systems built along these lines are said to have local memory. Before any computation can be carried out data must be distributed to the appropriate processing nodes. During the course of computation it is generally necessary for nodes to get information from other nodes memories. This is done by sending a message asking for the

information to be sent back in the form of a return message. If the two nodes concerned are not directly connected other nodes in the path between them can forward the message. Many schemes have been devised for connecting the nodes of such systems.

The systems designer of such distributed memory systems has several aims,

i) communication between nodes should be fast enough the handle the data flow among them,

ii) no node should have to support too many connections, and,

iii) the topology of the connections should somehow match the natural geometry of the problem to be solved.

The Network Structures:

The simplest interconnection system is a ring in which each node is connected to two others and the line of connections forms a circle. An extension of this is the mesh in which each node is connected to its four nearest neighbors. These topologies ensure only a small number of connections need be supported by each processor but have the drawback that a large number of nodes may be involved in sending a message between two particular nodes. This can result in long delays with the node waiting for data sitting idle, thus wasting computing time. This problem is alleviated in the slightly more elaborate Hypercube scheme. Using this approach nodes are connected as they would be if they lay at the corners of a multidimensional cube. In general a p-dimensional cube can connect nodes. Commercial systems are in use today which link nodes in such a Hypercube topology.

Hierarchical Communication Networks:

A more common approach is to design a communication network, where groups of processors are connected to separate network controllers which are themselves connected in a hierarchical manner so each individual

network controller is responsible for a manageable number of processing nodes. This type of network has the disadvantage that it increases the time for memory accesses since all requests must traverse stages of the network to reach the memory unit. This introduces variable length delay times for accessing data residing on different processing nodes. Also at times of peak data transfer activity the network can saturate resulting in long delays and degradation of system performance. Various types of networks have been designed to minimize congestion whilst keeping cost and speed to reasonable levels.

Hybrid Systems:

Parallel computing systems which combine the strengths of shared memory with local memory systems are classified as Hybrid systems. An example of such systems might consist of a number of nodes, each node consists of a modest number of processors that share a memory bank.

The nodes are then connected by a distributed memory system method. Alternatively a number of nodes which each can have their own local memory can also share a common memory.

Distributed Memory and Message Passing systems

Machine	Year	Topology	MaxNodes	CPU	MIPS[1]	Max Mem/node
Waterloop/64	1983	Loop	64	8086/87	0.03	128K
Cosmic Cube	1983	Hypercube	64	8086/87	0.03	128K
Mark II	1985	Hypercube	64	80286/287	0.04	256K
iPSC	1985	Hypercube	128	80286/287	0.04	512K
System 14	1985	Hupercube	256	80286/287	0.04	256K
NCube/ten[2]	1986	Hypercube	1,024	Custom	0.3-0.5	128K
Computing surface	1986	2-dimensional mesh	84	Transputer	N/A	128K
iPSC-VX[3]	1986	Hypercube	64	Vector	6-20	1.5K
FPS T-series[4]	1986	Mod. Hypercube	16,384	Vector	16-20	1.0K
Connection m/c[5]	1986	Hypercube	65,536	Custom	N/A	0.5K
Butterfly	1986	Banyon Switch	256	68020/81	0.1	1.0K
Mark III[6]	1986	Hypercube	1,024	Vector	20	4.0K

Comments:

1	MIPS - millions of instructions per second.
2	NCube has developed a single-chip node CPU that incorporates a 32-bit processor element, 11 bidirectional communications channels and a memory controller.
3	Intel VX series uses the iPSC's 80286/80287 node CPU as a communications control processor in conjunction with a microprogrammed vector function processor build around Analog Devices 3210/3220 floating point arithmetic units.
4	The T-series uses an Inmos Transputer (IMS T414) as a communications control processor in conjunction with a vector processor of the company's own design, using Weitek 1164/1165 Floating point arithmetic units.
5	The Connection machine uses a unique 1 bit serial processor, 16 of which are integrated into each physical processor chip.
6	The Mark III has a 68020/68081 node CPU and a vector processor of the company's own design, built around Weitek 1164/1165 floating point arithmetic units.

3.1.4 Data flow Architectures

The ability of large-scale integration (LSI) technology to inexpensively produce large scale numbers of identical complex devices, has made it possible to construct general purpose computers comprising of hundreds, perhaps thousands of asynchronously operating processors. Within such a computing system each processor accepts and performs a small task generated by the program, produces partial results, and sends these results on to other processors in the system. Many processors thus cooperate, asynchronously, to complete the overall computation. The data is fed by the program manager to the execution units. There is no local program on each unit to manage the execution but a manager will feed the data at the appropriate time.

The fundamental feature of dataflow architectures is an execution arrangement in which instructions are enabled for execution as soon as all of their operands become available.

The dataflow model:

The basic principles of dataflow semantics have been in existence for some time:

1- A dataflow operation executes when and only when all required operands become available (asynchronous).

2- A dataflow operation is purely functional and produces no side effects.

Operationally, these semantics are implied by the following model:
A *dataflow* program graph, FIGURE 10, is a directed graph where each node (presented in the figures by circles) is an operator with arrows connecting an output port to an input port of another node, provided that no two outputs are connected to the same input.

Dataflow problem presentation:

In a dataflow system a problem is presented as a directed graph (A complete chapter will be dedicated to graph theory) which is a collection of nodes connected by arcs. The nodes represent mathematical or logical operations and the arcs represent the flow of data from one operation to another. In a real systems nodes are represented as memory-processor pairs and the arcs by physical connections between processors.

The following are three examples of dataflow programs. These examples represent different forms of the number of inputs, outputs and operations.

Example 1:

This equation has both constant and variable operators. It has two variable inputs and produces only one output result. FIGURE 10 shows the equation with three arithmetic operations.

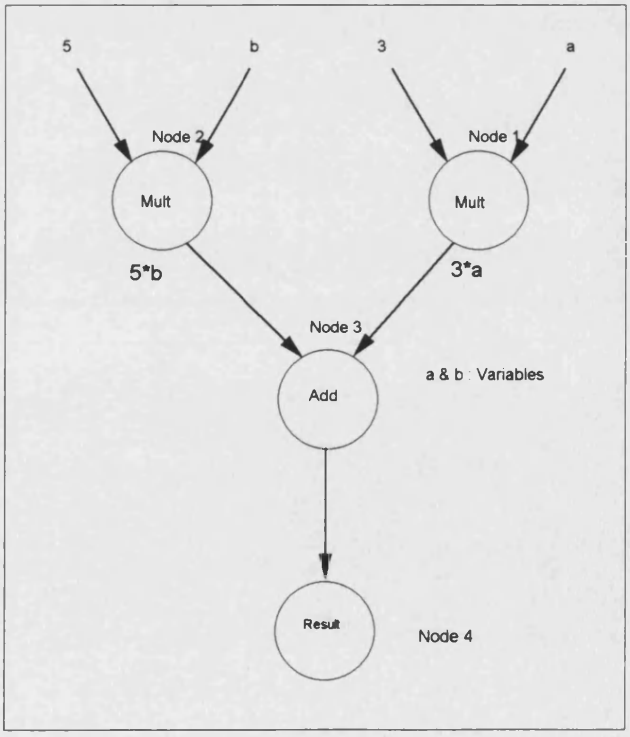


Figure 10: A simple dataflow program fragment.

Example 2:

This equation has two input variable operators. It produces two output results. See FIGURE 11.

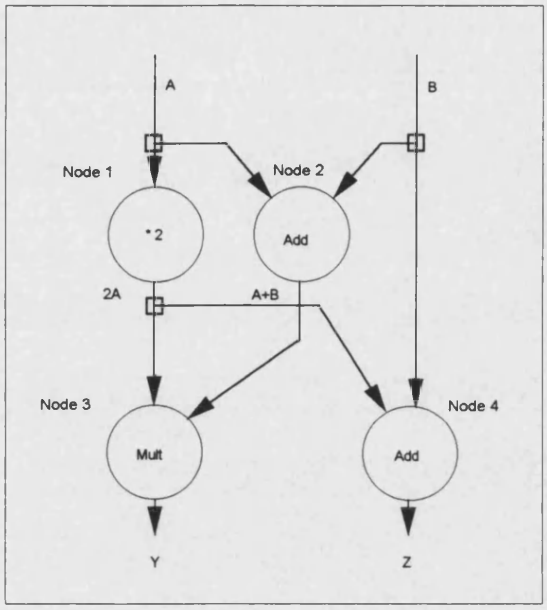


Figure 11: A second dataflow program fragment.

Example 3:

This equation depicts the solution of the quadratic equation. It has three variable inputs (A, B and C) and produces two output results (x_1 & x_2). See FIGURE 12.

$$X_1 = \frac{-B + \sqrt{B^2 - 4AC}}{2A} \quad \text{... Equ. 19}$$

$$X_2 = \frac{-B - \sqrt{B^2 - 4AC}}{2A} \quad \text{... Equ. 20}$$

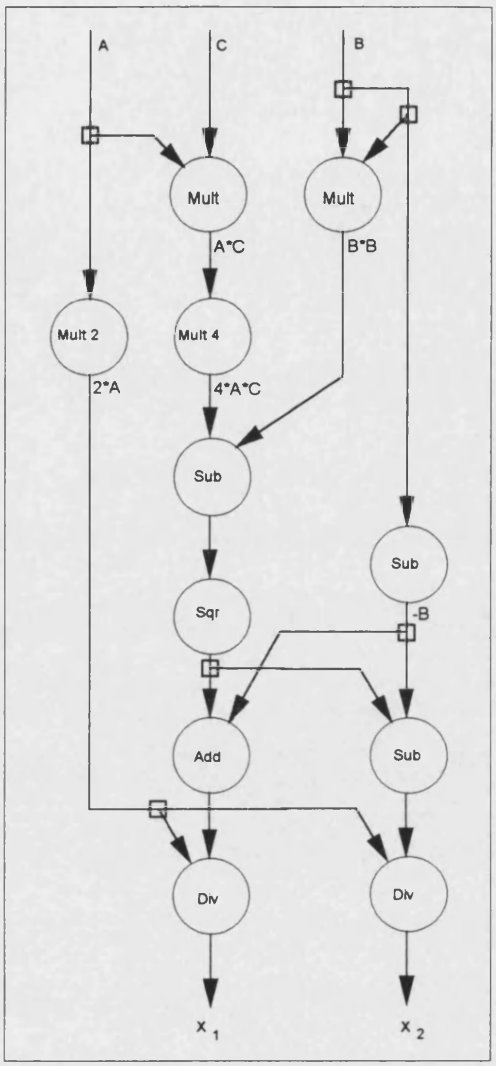


Figure 12: Dataflow program to compute quadratic roots.

Dataflow Execution:

A node in a dataflow machine does not execute its operation in a pre-determined sequence as specified in a program but instead waits until it has received all the data required to carry out its operations before executing the operation and passing the answer into the next nodes. A processor might wait for results from a number of processors and hand the result to a number of other processors.

In a dataflow system a very large number of nodes may all execute instructions simultaneously with no concern for what most of the other nodes are doing. Each node performs its appointed task only when the necessary data becomes valid. There is no danger of acting prematurely as with shared memory architectures. Also there is no danger of interfering with other processors say by trying to read a particular location at the same time as another processor. The two messages never try to pass along the same communications channel at the same time, thus avoiding the bottle-neck which occurs in parallel computing systems.

Dataflow Advantages:

One of the big advantages of a dataflow architecture is that it is much easier to program reliably than conventional parallel computers. A user merely needs to specify the operations to be carried out at each node without having to worry about the exact mechanics of the operation or the timing of the algorithm to be followed.

Dataflow Disadvantages:

1- One major problem with dataflow machines is the programming language used to program them. Conventional programming languages, which evolved from a basically Neumann paradigm, are not well matched to data flow architectures.

2- A fundamental problem with data flow and other highly parallel machines is that concurrency is limited by the communication network, which routes results from one processing node (PN) to another.

3.2 Execution of Parallel Algorithms

In theory, at best *n* processors working concurrently on a task must take *1/n* of the time that it takes for one processor to complete the same task. In reality, however, communication problems are amongst the factors which prevent this from being achieved. Speedup [Haynes-82] in the execution time of some algorithms can be increased by the addition of more processing modules. There are a number of factors that can prevent the realization of such a linear speed-up. They include the following:

Algorithm: The algorithm designed for a problem may not make full use of the parallelism which is available. The problem is usually split into a number of distinct modules then described by a mathematical model, these modules can be solved in parallel with data values interchanged. The algorithm may be inherently serial thus, an algorithm change needs to take place. Algorithms that possess a parallel segment will enhance their performance on parallel architectures.

Synchronization: Performance can be lost if the algorithm requires the processors to be synchronised at any point during processing. One example might require all processors to start a pass through the algorithm with the same data. This causes processors that could continue working to remain idle until all other units catch up. Another example might be where data has to be passed between the modules, resulting in some processors waiting and being idle while others reach the synchronization point.

Contention: Multiple processors competing for the same resource (shared variable, attention of a server, etc.) can slow down the execution of individual processors. Bus contention has a critical effect on the execution of parallel programs.

Overhead: A parallel algorithm might require more steps than its serial counterpart to solve the same problem. This overhead is the cost of managing parallelism.

Input/Output: Often, so much attention is placed on high processor - memory performance that Input/Output becomes an afterthought. Traditional Input/Output structures cannot feed a high-performance parallel processor fast enough to avoid processor idle time. Input/Output for applications measured on Illiac IV [Riaganati-84] has added a factor of up to 1.5 of the processing time to the total computation time.

When writing a parallel program one needs to consider both the algorithm to be implemented and the multiprocessor being used: for a given parallel algorithm, the same program coding is typically not appropriate for different multiprocessors. In fact, the algorithm itself might have to be modified depending on the multiprocessor being used. Such modifications arise from a number of software and/or hardware differences, examples of which are: communication architecture, Operating System (**OS**) calls and interprocessor data movements.

To exploit parallelism fully we must distribute the computation as much as possible, among the processing nodes. This we will generate a very large amount of interprocessor communication. The communication bottle-necks make it difficult for architectures based on global communication to exploit massive parallelism.

3.3 Conclusion

Parallel processing may ultimately offer a cheap solution for high speed computing, but this solution incorporates a number of limitations. The first arises from the time and effort required to program multiprocessor systems or adapting software used on uniprocessors to run on the multiprocessor. Real-

time considerations influence the choice of a system. The second limiting factor is the cost of such multiprocessor systems. In this chapter we have discussed a number of available parallel computing architectures

Selection of suitable Architecture:

The shared memory architecture shows an improved performance over the distributed memory system, Owing to the wide bus bandwidth provided by the former. Special hardware peripherals dedicated to arithmetic operations may be included within the system to improve its computational powers, and special hardware subsystems to improve the communications and reduce the communication costs.

The Dataflow computer:

The concept of dataflow is a useful tool to describe the operations of any algorithm. Numerical algorithms can be broken down to simple tasks. The execution of the tasks will depend on the availability of operands. Each task can be executed only if the operands are ready.

We can conclude that when using a dataflow system, the number of processors play a very important role. A large number of processors needs to be chosen so that the load is spread equally amongst all the processors. The reduction of communication overheads remains a target in such systems.

System enhancements capabilities

It is also desirable to equip the processing nodes with special hardware to enhance their operation. Such enhancements should effect both the areas of computation and communication. The computation power of a processing node dealing with numerical algorithms, such as ICCG, will significantly by the addition of dedicated vector processing hardware. The communication speed to transfer data among the processing nodes will

benefit from a special hardware. Both One-to-All and Caching mechanism introduce great advantages to system.

3.4 References

[Duncan-90] Duncan, Ralph, "A survey of Parallel Computers Architectures", *IEEE Computer*, Volume 23, February 1990, pp. 5-16.

[El-Ghajiji- El-Ghajiji, 90] Otman A., "Comparison of parallel computing architecture for real time diesel engine simulation", University of Bath, School of Electronic and Electrical engineering, M. Sc. thesis, 1990.

[Fathi-83] Fathi, Eli and Moshe Krieger, "Multiple Microprocessor systems: What, Why and When", *IEEE Computer*, Year 1983, Volume 16, March, pp. 23-32.

[Gimarac-87] Gimarc, Charles E., "A survey of RISC processors & computers of the mid-1980's", *IEEE Computer*, Year 1987, Volume 20, September, pp. 59-69.

[Haynes-82] Haynes, L. S., Richard Lau, Daniel Siewiorek and David Mizel, "A Survey of highly parallel computing", *IEEE Computer*, Year 1982, Volume 15, January, pp. 9-23.

[Leinrock-85] Leinrock, Leonard K., "Distributed systems", *IEEE Computer*, Year 1985, Volume 18, November 1985, pp. 1200-1213.

[Riganati-84] Riganati, John and Paul Schneck, "Supercomputing", *IEEE Computer*, Year 1984, Volume 17, October, pp. 97-113.

[Rumbaugh-77] Rumbaugh, James, "A Data Flow multiprocessor", IEEE transactions on *computers*, vol. C-26, number 2, February 1977, pp. 138-146.

Chapter 4

Simulation of Parallel Computers

4.0 Simulation of Parallel Computers

In the previous chapter, we discussed the various aspects of multiprocessor architectures. The problem of system degradation in performance, factors affecting speedup results, different architectures and examples of the dataflow models. We will observe in this chapter the basic behind the software systems using in simulating computer systems. Previous attempts by a number of researchers are summarized and categorized for reference.

Introduction

Simulations mimic a physical process to generate performance estimates, identify errors and verify correctness before designers fabricate a prototype. Digital hardware designs, industrial control circuits, and aircraft are usually simulated extensively. This chapter, titled Simulation of Parallel Computers, discusses the concept of Discrete event simulation, which forms the basis of computer simulators and surveys previous attempts of simulating processors, uni- and multiprocessor architectures. We have proposed a classification that combines the different forms of simulators. We have also prepared a comparison table to show a summary of past research in this field.

4.1 Discrete Event Simulation

A discrete event simulation model assumes the system being simulated only changes state at discrete points in simulated time. The simulation model jumps from state to state upon occurrence of an *event.* The internal operation of a digital computing system can be efficiently simulated using discrete event simulation. The internal state transitions resulting from the execution of the instruction are modeled around the clock cycle.

Computer architecture simulators are programs used to simulate the operation of a computer system. Simulators are grouped according to their computer environment into *Sequential* and *Distributed* simulation programs. Sequential simulators run on a single processor, whereas the Parallel simulators run distributed among a number of different processing nodes. Sequential simulators typically utilize three data structures:

1- The state variables that describe the state of the system. These state variables describe the state of the system during the different simulation phases.

2- An event list containing all pending events that have been scheduled, but have not yet taken effect. This event list is represented by a queue model. The simulator may utilize a number of queues to organize the simulation and its internal operation.

3- A global clock variable to denote how far the simulation has progressed. This variable increments in discrete values. The system under simulation will take the actions needed at each clock change. All program modules are tested for the effect of the clock change. Thus, the simulator needs to scan all processing modules during each clock cycle to update the new state of the system.

4.2 Discrete Event Models:

Systems can be described as discrete or continuous depending on its internal operation and status change. In *discrete* event simulation models the variables are discrete quantities representing states of entities in the system. Interactions between entities only take place at discrete points in time only, separated by intervals of inactivity. Such interactions are usually known as events. Time is advanced usually from each event to the next. At each step, all system changes implicit in current events are made and any new events

called are added to a future events list. The actual timing of events is usually affected by stochastic factors. Discrete event models are applicable to wide range of problems concerned with systems of moving units. They have been used to model neutron flow in nuclear reactors, transport systems and interaction of combat units on the battle field. We will use this models to simulate a multiprocessor computing system.

Continuous change models can conveniently be run on analogue computers, if high accuracy is not required. Fixed-point and discrete event models, on the other hand, dealing as they do with discrete changes in time, are more suited to the digital computer.

Methods of study:

Two methods are typically used for multiprocessor studies: *analytical* and *simulation* modeling. Analytical models become rigid when multiprocessor dynamics are considered and are not practical for application-dependent studies. Simulation, with the correct assumptions, is a feasible approach that can produce an accurate picture of the dynamic behavior of multiprocessor systems. Different algorithms can be modeled and simulation results obtained.

We have opted for the *simulation* option in this research, as the analytical modeling will not serve our needs. The algorithm analyzed in this project, namely the Backsubstitution, has same structure but the distribution of matrix elements is different every time and depends on the CAD model used. The matrix structure introduces patterns which may improve the parallel simulation of the ICCG.

Simulating Multiprocessor systems:

Parallel computing architectures are digital systems that state transitions take place during the clock change. These digital systems are best simulated using the DES models. Creating different models to describe the

behavior of the computer subsystem and combining them in one system will result in a model for parallel architectures. Simulation of the computer system is a tool for obtaining a meaningful quantitative measure of the performance of a proposed system design.

Purpose of Computer Simulators:

Computer simulators are designed and built to serve a number of purposes. The following list names some of the purposes for simulator usage:

1- To aid computer designers in developing high-performance systems by testing different architectures and designs.
2- To help researchers in testing predictions about system behavior and performance.
3- To facilitate debugging and testing of programs which are expected to run on parallel processing systems.
4- Test applicability of algorithms for specific hardware architectures. If the same algorithm is run on a simulator under different hardware settings each time, the performance results can show its behavior.

4.3 Simulation System Goals

Simulation of computing systems at an architectural level, with an appropriate abstraction, can offer an effective way to study critical design choices. Estimation results will be a valuable aid if the following **conditions** are met in the design and implementation of the simulation software:

1- The simulator includes details of different parallel architectures (e.g., shared- or distributed memory systems). This will allow the acquisition of results for different forms of architecture. These results may be used to facilitate comparisons.
2- The ability to model the algorithm that will eventually run on an actual system in detail, to allow investigation of different forms and

decomposition factors. This would allow the programmer a useful insight to the problem.

3- The performance of the simulator is adequate to examine designs executing significant real code.

A simulation system with these goals is described in the next chapters of this thesis together with our approach used in its implementation. Its application to the study of a particular class of numerical algorithms, namely ICCG Incomplete Choleski Conjugate Gradients was described in chapter (2) and the results will be later demonstrated in chapter (7).

The **design goals** that emerged during the analysis phase of this research project were as follows:

1- That the simulation system should support flexibility with regard to simulated system structure and interconnections.

2- That, in order to accomplish runs in acceptable elapsed time, the details of simulation should be particularly focused on the details of the processing node activities, communications, process scheduling and special hardware features of the simulated system.

3- To produce performance estimates for different hardware configurations running the ICCG algorithm.

4.4 Performance Estimation

Performance estimation of parallel programs plays an important role in the determination of the suitability of a given computer architecture in performing the algorithm tasks, Thus, performance estimation is important for two reasons: to achieve a more efficient systems design, and to determine the capabilities of the system once it is designed without physically implementing it. These requirements call for a tool which will allow the designer to model the computing system in enough detail to know what is

happening in the operation of the system under the design stage. The designer can measure the effects of changes in the design and can estimate the performance achievable with the system for specific applications.

Level of simulation abstraction

Multiprocessors, as any other digital hardware system, can be described at various levels of abstraction. Simulation of such systems may present either the functional behavior of all the system or the gate level behavior of all subsystems within the multiprocessor. Functional descriptions merely capture the behavior of the system without revealing details of the internal operations. On the other hand the gate level simulation is more robust and includes great amounts of detail about the digital system. Functional models simulate faster than gate level models. However, the trade-off is that the simulation results of the functional model are not as accurate (primarily from the timing standpoint) as those from the gate level models.

Why we use computer simulators?

One of the difficulties with many multiprocessor computer designs is obtaining a meaningful measure of the performance of the system at an early stage. Many factors can affect performance, so even straight forward changes in a system do not have the expected effects. For example, one might expect that doubling the clock speed of a processor within the processing node and upgrading all related components would double the speed of processing. But if this change is implemented with no other adjustments to the system, the increased processor speed might cause the processor to access the memory bus frequently, thus reducing the amount of time available to other resources (e.g., processing nodes, I/O, etc.) using the bus and bus contention. For applications which require large amounts of data and I/O transfers, this conflict could limit the performance of the revised system to a fraction of the expected gain.

Computer Simulators Implementation:

A number of techniques are used to simulate the operation of distributed computers in general. These simulation programs can be *serial* or *parallel*. The implementation of the simulator itself may take one of two forms of implementation, either a serial program which will run on a uniprocessor or a parallel implementation which will run on a network of processors.

FIGURE 13 shows the relationship between the different parts forming the computer simulation system. These parts are:

1- The program or algorithm describing the tasks which need simulation.

2- The architecture of the computing system to be simulated.

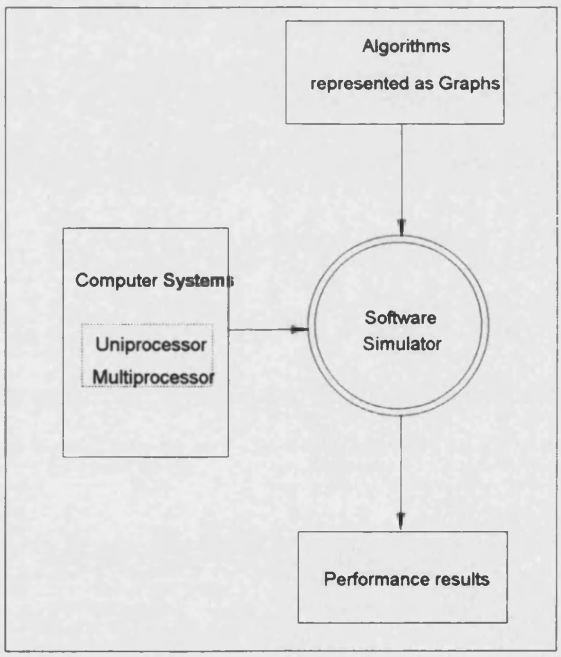


Figure 13: Software simulation of underlying architectures.

4.5 Previous Attempts

The simulation of multiprocessor architectures and subsystems implemented in the past will be described briefly in the following sections.

A review of the technical literature has led us to the following (proposed) classification of simulator types:

1- Processor Instruction Simulators.

2- H.L.L. Source Code Simulators.

3- Operating system Simulators.

4- Stand-alone Simulators.

first summarizes the above categories, and gives the reference information of the specified research.

4.5.1 Hardware subsystems

This section discusses a group of simulators that is related to the hardware architecture of the parallel computer.

The **K9** developed by Beadle [Beadle-89] is an example of a simulator of distributed-memory parallel processors. K9 is written in C++ [Stroustrup-87] and runs on Sequent Symmetry. Application code for K9 can be written either in C++ or C. It provides a fast simulation of DMPP (Distributed Memory Parallel Processing) allowing algorithms and architectures to be matched before a distributed hardware platform is developed. **K9** allows multiple architectures and processors to be evaluated for a mixture of algorithms, with real data sets.

ConLab (Concurrent Laboratory) developed by Jacobson [Jacobson-90] is an environment for developing algorithms for parallel computers. It is an interactive environment in which one can simulate MIMD architecture with distributed memory and communication with message passing, as well as,

MIMD architecture with shared memory. The ConLab is written in C and runs on a Unix-based system.

The simulation of bus architectures for multiprocessor systems is considered by Wear [Wear-82], who presents the results of the simulation of three different bus architectures used in multiprocessor systems. The timeshared bus, multibus, and cross bar switch configurations are modeled. This paper presents the results in graphical form, and does not detail the mathematical model.

Oryuksel [Oryuksel-89] presents a simulation model for performance analysis of multiple-bus multiprocessor systems with shared memories.

4.5.2 Processor Instruction Simulators

Simulation of a single processor within a multiprocessor computing system has been considered by a number of different authors. This type of simulator is restricted to the processor being simulated and it is not flexible enough to simulate other types or configurations.

The simulation of a microprocessor (namely the 6502) is carried out by Chang [Chang-87].

Kreulen [Kreulen-90] used the MC68020 as the processor model for microprocessor-based simulation. Both Gorissen [Gorissen-85] and O'Grady [O'Grady-87] have based their simulations on the Intel 8086 microprocessor and the Intel 8087 numerical coprocessor. Both systems are modeled on the instruction level. The 8086 has about 130 instructions. Each of these instructions are simulated separately. The instruction set of the processor including flag handling is completely reproduced according the specification of the microprocessor.

Gorrissen [Gorrissen-85] presents a simulation different to that of Beadle [Beadle-89]. The multicomputer simulation presented by Gorrissen [Gorrissen-85] simulates in detail the hardware functions of the system. The simulation is based on an architecture that has a Time-Shared Bus (TSB) bus and various processor and memory parts. Each processing node can access not only a local memory but also a global memory. The simulation software executes the instructions of the microprocessor in detail, thus affecting all of its components, local and global memory and also the system bus. For the execution of real programs on the simulation model [Gorrissen-85] has modeled the system on the microprocessor instruction level.

4.5.3 H.L.L. Source Code Simulators

This group of simulators accepts the source code of the programs to be simulated in High Level Language (H.L.L.) form. The source code is written in a computer high Level Language. An example of this group will be [Beadle-89].

4.5.4 Operating system Simulators

The simulation of software structures for different parts of the operating system is also considered. Simulation models for Multitasking by Karatza [Karatza-87] and [Karatza-88]; also, the simulation of dynamic task allocation in a distributed computer system is discussed by Andert [Andert-87].

Other types of related simulations deal with different aspects. One example is Shaw [Shaw-87] where a simulation of a parallel processor with unbalanced loads is considered.

4.5.5 Stand-alone Simulators

The Universal simulator SIGMUS developed by Schmidt [Schmidt-81] represents a simulator, where the simulation of parallel processes of any type and simulation of system models of any abstraction level are claimed to be possible. Other simulators have been designed for specific and microprocessors. One of the main contributors is Butler [Butler-86-1] and [Butler-86-2] in which the EUCLID simulator is discussed in detail.

Characteristic	Groups	Examples
Implementation	Parallel	[Li-90]
	Serial	[Bead-89] [Krue-90] [Butl-86]
Microprocessor	8-bit micro 16-bit micro	6502 [Chang-87] 8086 [Gorissen] MC68020 [Kreulen-90]
Bus Structre	Single bus	[wear-82]
	Multiple bus	[Oryu-89]
Architecture Simulated	Distributed Memory	[Schm-81] [Jaco-90] [Butl-86]
	User-defined topology	[Schm-81] [Bead-90] [Butl-86]
	Shared Memory	[Schm-81]
Form of Input	Low Level	[Krue-90]
	High Level	[Bead-89]
	Special	[Butl-86]
Form of Output	Tables	[Kreu-90]
	Graphics	[Bead-89] [Butl-86]
Environment	Interactive/Window	No example found.
	Menu	No example found.
	Data Files	[Krue-90] [Butl-86]
Scheduling	User supplied	[Kreu-90] [Butl-86]
	Automatic	No example found.

Table 1: Comparison Table of Different *Simulators.*

Li [Li-90] describes a simulation tool called the Simulated Parallel Architectures in a Distributed Environment (SPADE). SPADE is implemented on a network of workstations with Unix. With SPADE, a user can experiment with various existing and hypothetical parallel machine models. SPADE is an example of distributed simulators.

4.6 Discussion and Conclusion

This chapter has introduced the concept of simulation, and identified the DES concept as the build block for any digital system simulator. DES is the most suitable form for simulation of parallel computing systems. The survey of previous published papers and articles revealed that a number of topics need to be investigated further. Table (4.1) groups the different categories of software simulation packages encountered. We can conclude that there is further scope to develop a multiprocessor simulator with scheduling capabilities.

In writing a simulator for parallel and distributed computing system, the 3 data structures described in this chapter will form the basic skeleton of the proposed computer simulator. After reviewing a number of different computing architecture simulators, we have concluded that we will develop our own simulator with the capabilities to both schedule and simulate computer tasks.

The design and implementation of the proposed computer simulator will satisfy the goals discussed in section (4.3). Mapping the ICCG algorithm, discussed in detail in chapter 2, onto a user defined architecture, incorporating communication enhancements techniques, will produce useful performance measurements of the algorithm. This simulator, which will be equipped with an option to allow intelligent scheduling of tasks, will be used to investigate the effect of the sparsity structure on the performance of parallel execution of the ICCG algorithm.

4.7 References

[Elderdge-90] Eldredge, David L., John D. McGregor and Marguerite K. Summers, "Applying the object-oriented paradigm to discrete event simulations using the C++ language", February 1990, *Simulation*, pp. 83-91.

[Stroustrup-87] Stroustrup, Bjarne, "The C++ programming language", 1987, Addison-Wesley, ISBN 0-201-12078-X.

[Andert-87] Andert, Ed, "A simulation of dynamic task allocation in a distributed computer system", *Proceedings of the 1987 Winter Simulation Conference*, pp. 768-776.

[Beadle-89] Beadle, Peter, "**K9**: A simulation of distributed-memory Parallel processor", Proceedings of the Supercomputing 1989, pp. 765-774.

[Butler-86-1] Butler, James M. and A. Y. Oruc, "EUCLID: An architectural multiprocessor simulator", *Proceedings of the 6th international conference on distributed computer system*, Boston 1986, USA, pp. 280-286.

[Butler-86-2] Butler, James M. Butler and A. Y. Oruc, "A Facility for Simulating Multiprocessors", *IEEE Micro*, Volume 6, Number 5, October 1986, pp. 32-44.

[Chang-87] Chang, Chi-Keng and Kuo-An Hwang, "A Computer simulator for concurrent processors", Journal of Chinese institute of engineers, Vol. 10, No. 4, July 1987, pp. 447-452.

[Fujimoto-90] Fujimoto, Richard M., "Parallel Discrete Event Simulation", *Communications of the ACM*, October 1990, Vol. 33, No. 10, pp. 30-51.

[Gorissen-85] Gorissen, Jack and Karl Lebsanft, "Simulation of a multicomputer system on instruction level", Proceedings of the 1985 summer computer simulation conference, 1985, pp. 118-122.

[Jaconson-90] Jacobson, Peter, "The ConLab environment", report UMINF-173.90, ISSN 0248-0542, University of UMEA, institute of information processing, Sweden.

[Karatza-87] Karatza, Helen D., "A Simulation model of multitasking in parallel processing", *International Journal of Modeling & Simulation*, USA, Vol. 7, No. 1, 1987, pp. 37-42.

[Karatza-88] Karatza, Helen D., "Simulation models for parallel processing with programs", *International Journal of Modeling & Simulation*, USA, Vol. 8, No. 3, 1988, pp. 78-82.

[Kreulen-90] Kreulen, Jeffrey T. and Matthew J. Thazhuthaveetil, "Application - dependent simulation of microprocessor - based multiprocessors", *Microprocessors and Microsystems*, Volume 14, Number 7, September 1990, pp. 467-473.

[Li-90] Li, Xiaobo and Yian-Leng Chang, "Simulating Parallel Architectures in a distributed Environment", Journal of Parallel and Distributed Computing, Volume 9, Year 1990, pp. 218-223.

[Marsan-82] Marsan, M. and Mario Gerla, "Markov Models for multiple Bus Multiprocessors Systems", IEEE *Transactions on Computers*, Vol. C-31, No. 3, March 1982, pp. 239-248.

[Marsden-84] Marsden, B. W., "A standard PASCAL Event Simulation Package", Software - Practice and Experience, Vol. 14(7), pp. 659-684, July 1984.

[O'Grady-87] O'Grady, E. Pearse and Chung-Hsien Wang, "Performance limitations in parallel processor simulations", Transaction of the society for computer simulation, October 1987, Vol. 4, No 4, pp. 311-330.

[Onyuksel-89] Onyuksel, Ibrahim and K. Irani, "Simulation experiments for performance analysis of multiple-bus multiprocessor systems with nonexponential service time", *Simulation*, January 1989, pp. 18-23.

[Schmidt-81] Schmidt, Klaus and Helmuth Schmidt, "The Universal Simulator SIGMUS" proceedings of the summer simulation conference, Washington D.C., 1981, pp. 631-635.

[Shaw-87] Shaw, Wade H. and Timothy S. Moore, "A simulation study of a parallel processor with unbalanced loads", *Proceedings of the 1987 Winter Simulation Conference*, pp. 759-776.

[Wear-82] Wear, Larry L., "A simulation of bus architecture for multiprocessor systems", *Proceedings of the 1982 winter simulation conference*, pp. 268-278.

[Zehendner-89] Zehendner, E. and Th. Ungerer, "A simulation Method for parallel computer architectures", *Microprocessing and Microprogr*amming, 28, (1989) pp. 209-212.

Part THREE

Scheduling and Simulating

This part of the thesis is dedicated to our method in scheduling and simulating the parallel tasks and architectures.

Chapter 5

Critical Path Analysis and Scheduling

5.0 Critical Path Method and Scheduling

This chapter introduces the Critical Path Method (CPM) concept. It shows how to build a network graph, compute the critical path for a given graph network. It also identifies the use of CPM in the multiprocessing environment. Scheduling of activities and processor assignment is also discussed. The goals and structure of our simulation program is given at the end of this chapter.

Introduction

It is our objectives to investigate the execution of the Incomplete Choleski Conjugate Gradient algorithm on parallel computing architectures, thus the problem of obtaining the maximum speedup is of great importance. The CPM provides a strategy by which we can identify the path needed to complete the execution o the network that will result in the least time possible.

5.1 Introduction to Critical Path Method

Critical Path Method (**CPM**) is a planning methodology which represents each task in a directed graph of activities. The CPM is commonly used in operational research for planning and scheduling. Other applications of CPM is in process control engineering and identification of shortest traveling distances. It is also used as a scheduling tool in distributed programs.

Each activity network is a hierarchy of tasks, a high-level task being comprised of a set of lesser tasks. Each node in the graph represents a particular task. Edges point from prerequisites to the tasks which depend on them. A task cannot be started before all of its prerequisites are finished.

Given an estimate of how long each task in the network will take, CPM determines the time by which each task must start and be finished such that the deadline will be met.

5.2 The Network Diagram

Diagrams like that shown in FIGURE 14 are also called networks. The network has one initial node and one terminal node. The circles in FIGURE 14 are numbered and represent the nodes. Associated with each node is an activity called Task. The lines represent data movement. Each line has an arrow indicating its direction. The network as a whole shows a series of activities that must be performed to complete the project. The arrows show which activities and nodes logically precede others. An event that results from completion of more than one task is called a merge event; an event that represents the joint beginning of more than one activity is called a burst event. Before any activity can start, all preceding activities must be completed (but not simultaneously). An arrow's length and its compass direction are insignificant. Nodes are sequential processes that execute in parallel and communicate only by exchanging messages.

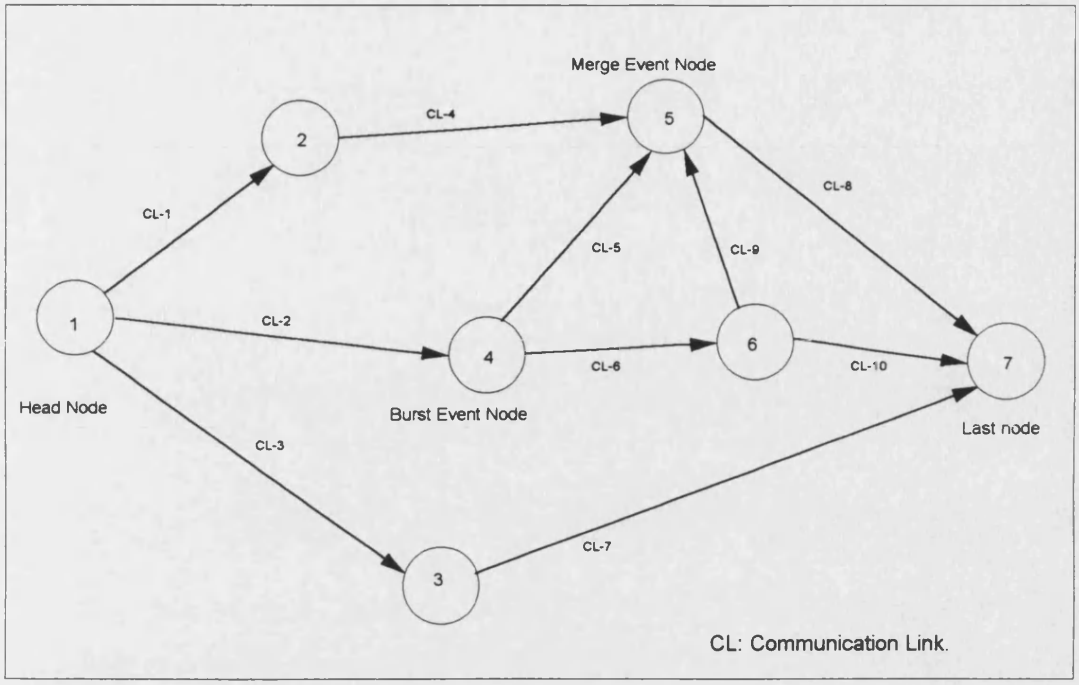


Figure 14: A network of connected Nodes.

5.3 Identification of Critical Path

The network is a graph structure in which any task has a number of predecessors and one or more successors. The *head_node* has no predecessors. Two terms need defining at this point, the first is Earliest Start Time by which an activity (task) can occur, within the logical and imposed restraints of the network. The second is Latest Start Time by which an activity (task) can start within the logical or imposed constraints of the network without affecting the total project time duration.

Within a network of this sort the critical method can be expressed by the following statements:

1- A task's latest end is equal to its successor's latest start.

2- A task's earliest end is equal to its successor's earliest start.

These two statements dictate two sweeps of the activity network. One sweep or pass determines the earliest time each task can start and end. The

other determines the latest time each task can start and end. Working backwards from the network latest end of each task can be determined, as can its latest start.

The first sweep or pass over the whole network will be called the *forward_pass()*. In this pass the network is swept from the *head_node* to the *end_node*. Each node in the network is checked and its links are assigned the *early_start_time*. The second sweep or pass will be called the *reverse_pass()*. In this pass the network is swept starting from the *end_node* and proceeding backward to the *head_node*. Each node in the second pass is checked and all its links are assigned the *latest_start_time*. Any node that has at least one link with equal earliest and latest start times will classified as a *critical* node. By the end of the second sweep, a list of critical nodes that identify the critical path is produced.

Let us consider the following example which shows how to identify the *early_start_time* and the *latest_start_time* for the given network of 7 nodes. The number above the arrow represent the time needed to move data in the direction specified by the arrow. FIGURE 15 shows the original network.

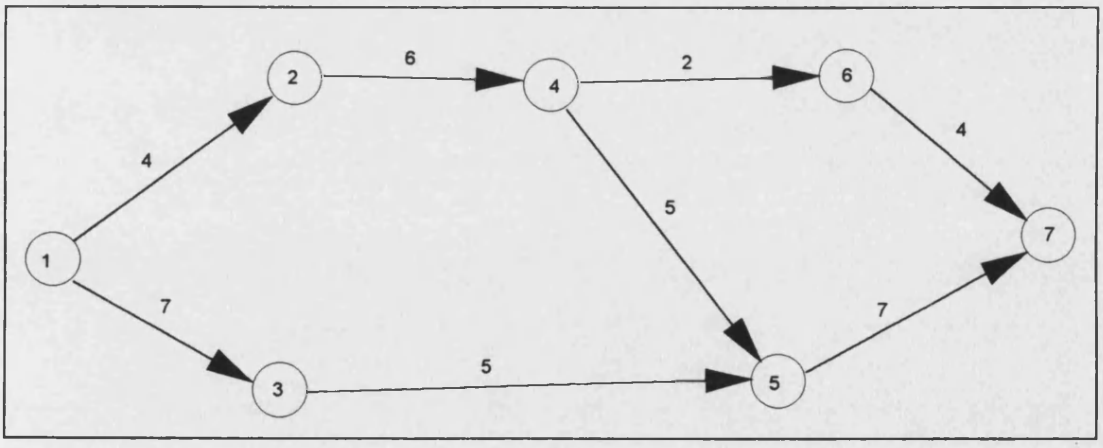


Figure 15: Original network.

In sixteenth the earliest start time is computed for each node. The sweep starts at node number 1 and visits each node until the end node number 7. The earliest start time for each node is assigned.

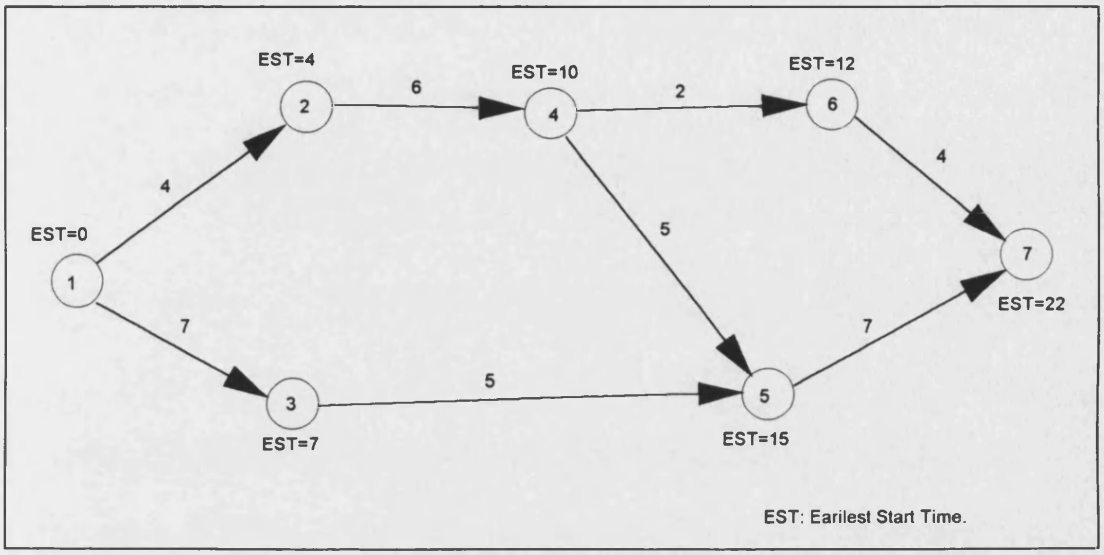


Figure 16: Forward sweep to compute the earliest start times.

Whereas in seventeenth the latest start time is to be computed for each node. The sweep starts at node number 7 and moves backwards towards node number 1. Each node visited will have its latest start time value assigned.

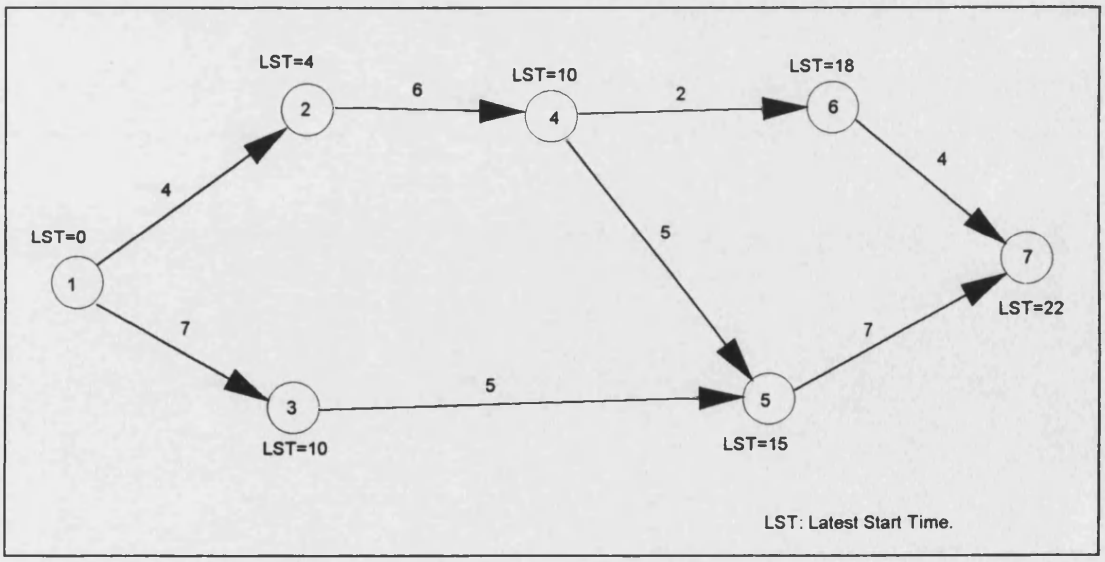


Figure 17: Reverse pass to compute the latest start times.

At this point of time the critical path can be identified. The critical path of the network is the sequence of activities that constitutes the longest time path through the network and thus determine the minimum expected time in which the network can be completed. The critical path can be traced to activities which have equal early and late start times. The activities residing on the critical path always have a difference of zero between the early and late start times.

The critical path for the network in fifteenth follows the following sequence of activities: 1- 2- 4- 5- and 7. Although activities 3 and 6 are expected to be completed by the start of activity 7, and still expect to complete the network on time.

Node	EST	LST	Status
1	0	0	Critical
2	4	4	Critical
3	7	10	Non-Critical
4	10	10	Critical
5	15	15	Critical
6	12	18	Non-Critical
7	22	22	Critical

Table 2: The link information (earliest and latest start times) for the network of 7 nodes.

5.4 Critical Path and Parallel Programs

In the execution of parallel programs, the all process completion time is an important performance measure. When completion time is used as the measure, speed is the major concern. One way to determine the cause of a program's long completion time is to find the event path in the execution history of the program that has the longest duration. This Critical Path identifies where in the program attention is needed.

Characteristics of distributed programs:

In order to identify the Critical Path CP in a distributed program, it should be seen as having the following characteristics:

a- It can be broken down into a number of separate activities. These activities must have comparable execution times.

b- The time required for each activity can be measured or estimated,

c- The interaction of activities should be known, thus allowing some activities to be executed serially (sequentially), while others may be carried out in parallel.

d- Each activity may require a combination of resources, e.g. CPU (integer, float, logical etc.), memory physical space (cache, local or remote) and I/O devices. There may be more than one feasible combination of resources for different activities, and each combination is likely to result in a different duration of execution.

Based on the above distributed program characteristics, we can use the critical path method (CPM) to analyze and schedule a program's execution. Previous attempts to utilize the CP were carried out by [Barry-85] [Kasahara-84] [Kohler-75] and [Yang-88].

5.5 Application of CPM to Parallel Programs

The CPM is used in the parallel programming environment in the *scheduling* parallel tasks or activities. The objectives of scheduling is to assign each of the nodes

$$N \{ N_1, N_2, \ldots, N_{nodes\_max} \} \text{ to one of the}$$
$$\text{PN processing nodes } \{PN_1, PN_2, \ldots, PN_{pn\_max}\}$$

Such that the minimum execution time will be obtained. This situation can be improved in a number of ways:

1- The programmer should have tools that provide parallel program performance information. These tools should be able to provide, at the very least, some of the more routine analyses that a programmer would need to have done.
2- The programmer must rely upon intuition to make changes to improve performance. Parallel programming is a very intuitive craft, and less intuitive programmers will have more difficulty improving programs. Guide-lines for program improvement would help to reduce the intuition needed and make parallel programming easier.

Scheduling the computation tasks to achieve minimum total execution time requires an understanding of the data dependencies that exist among problem variables and an estimate of times of individual tasks. The data dependency graph in fourth identifies the tasks and the data dependencies among variables computed during the execution of the backsubstitution algorithm during an iteration of ICCG. Matrix coefficients and constants are available at the start of the period of computation and appear in the left-side of the graph; the evaluated variables are at the bottom. Each node represents one or more arithmetic task. Arrow-pointed arcs identify a dependency relationship between two variables; the variable at the lower end of an arc (destination) depends on the variable at the upper end (source). Due to this dependency the computations associated with nodes connected by an arc cannot be performed in parallel because the variable needed to complete the destination task is not available until after completion of the source task; thus, arcs in the dependency graph network identify computations which must be performed sequentially. Computations associated with nodes that do not have a source-destination relationship can be performed in parallel.

The general problem of scheduling n interdependent tasks for minimum time execution on m processing nodes requires understanding both the network structure and the computing architecture used. However if the number of tasks is relatively small and the number of processors is sufficiently large, analysis of the dependency graph can assist in developing

an optimal schedule. Scheduling is also simplified if the time required to exchange data among processing nodes is assumed to be negligible; this assumption may be justified if the ratio of time required for data exchanges to time required for computation tasks is small. In this case it is helpful to identify the critical path which represents the longest serial computation sequence on the dependency graph. The critical path determines the lower bound on the computation time for performing the computations on the graph with parallel processing. Under no circumstances (i.e., no matter how many processors are used) is it possible to perform the computations on the graph in less time than that required for the computations on the critical path if the precedence relations are to be satisfied.

5.6 Mapping of Parallel Tasks

The objective of the mapping problem is to match the algorithm structure with the machine (hardware) structure. For a uniprocessor system it is easy to map an algorithm by using compiler-based tools. However, as the number of processors increases, the mapping problem becomes more difficult. A general approach to the mapping problem includes three steps:

1- Identification of parallelism within the algorithm,

2- Partitioning of the algorithm in to sequential tasks,

3- Scheduling and allocation of processing nodes for the algorithm tasks.

5.7 Scheduling of Parallel Tasks

In a parallel programming environment, scheduling may be **static** or **dynamic**. In static scheduling policy, the binding between processors and processes is done at compile time. This is also known as Single Static Assignment problem. Static scheduling has low runtime overheads but is less, and often results in poor load balancing. In dynamic scheduling policy, the time of binding of processors to processes is delayed until runtime. A process can be executed by any processor that becomes available, under

certain constraints. Dynamic scheduling needs more runtime overheads but also produces better load balancing and less interprocessor communications.

5.7.1 Rules of Scheduling:

The following rules will be applied to our simulation program:

1- A schedule must casual: if a *node1* produces a value that is used by *node2,* then *node1* must be scheduled and executed before *node2.*

2- Also, no two nodes are scheduled to the same processor at the same time slot.

3- If the node input operands are provided at the right time and made available the right place, the program will compute the desired results and provide them at the times and places given by the simulator. Data will arrive on time and will be processed according to the simulator instructions.

5.7.2 Optimal Schedule:

An optimal schedule can be determined as follows. Critical path computations are scheduled on one processor; then non-critical-path computations are scheduled on other processors. Considering the following assumptions:

1- When a non-critical path computation has to scheduled in order to have its result available when needed for critical path computation, thus not delaying the overall execution time of the network.

2- When a computation task can be scheduled with assurance that all its predecessors have been computed, and

3- At any instant which processors are available for assignment.

All schedules in which a separate processor performs all computation tasks on the critical path (and no other tasks) without delays are optimal schedules; here it is understood that processors performing non-critical path

computations have to complete them within the time interval required for critical-path computations.

If data exchange time is not negligible (as in multiprocessor simulations), data exchanges that occur during a period of computations must be included in the tasks considered by the scheduling algorithm. This makes scheduling more complex not only because the number of tasks to be scheduled increases, but also because the requirements for specific data exchanges depends on the schedule itself. For example, one schedule for utilizing 3 processing nodes might require the transfer of the result of computations to other processors, while a second schedule for utilizing 5 processing nodes might require only local use of the same result.

The choice of the scheduling method can have a dramatic influence on the parallel execution time of a program for the following reasons:

1- There is loss in parallelism when potentially parallel nodes are assigned to the same processing node. If these two nodes were executed in parallel a better performance would be expected.
2- Since all data transfers need a finite time to take place, this time need to be minimized to obtain better speedup results.
3- A good choice of the node sequence can often reduce the wait component and utilize the time in more useful functions.

5.7.3 Single Static Allocation

Data flow analysis techniques can be used to analyze and improve parallel program performance. In **static** data flow analysis, the program source code is analyzed to provide information about the program without having to execute it. In **dynamic** data flow analysis, the results of program execution are combined with the information from static data flow analysis to provide information on program performance.

The assignment of nodes to processing nodes PNs can have a large impact on the performance of the multiprocessor system. For example, since each PN is a sequential computing machine, nodes that potentially can execute in parallel cannot do so if they are assigned to the same PN. Performance can also be affected by data communication delays in the inter-processing communication network. It takes many more clock cycles to transmit data from one PN to another than it does to transmit data into the PN's memory (local write cycle), which by-passes the communication mechanism completely.

This leads to three goals for efficient allocation of nodes into PNs:

1- Minimize IPC by assigning nodes connected in the graph to the same PN.

2- Maximize the parallelism of the graph by assigning nodes that can execute in parallel to separate PNs.

3- Balance the computational load evenly between the PNs.

5.8 Scheduling the Backsubstitution Algorithm using Critical Path method

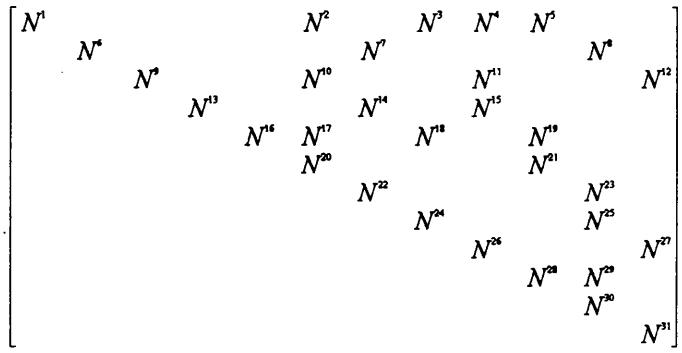
Let us now apply the CPM to the numerical class of algorithms. We have identified, in chapter 2, the backsubstitution algorithm to be the computation and communication bottleneck of the ICCG solution. In this section we will consider three matrix structures.

The following 3 matrix examples have the same matrix size but different sparsity structures. Only non-zero elements are shown. In Model 1 the matrix has 31 elements.

Model 1

Number of elements: 31

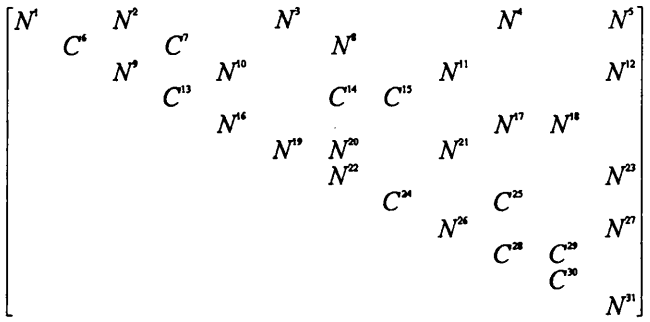
Number of Critical path elements: Not identified.



Model 2

Number of elements: 31

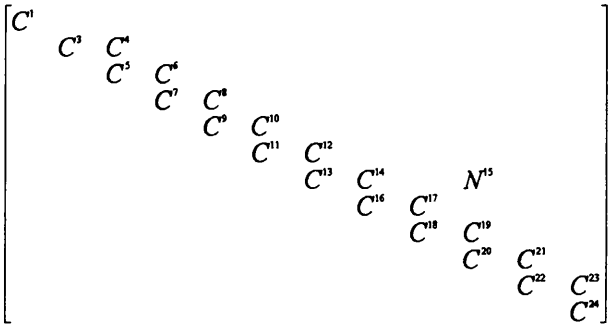
Number of Critical path elements: 10



Model 3

Number of non-zero elements: 24

Number of Critical path elements: 23



Model 2 has the same matrix size, but the length of the critical path is 10 nodes. In model 3, where the matrix structure is more dense on the diagonal elements, then the critical path is longer and hence of length 23 elements. In this model 23 elements out of the 24 reside on the critical path. Model 3, on one hand, can not lend itself to parallel execution due to the

inherent serial structure in the distribution of its elements, on the other hand, model 2 has some parallel chain of nodes that will improve its parallel execution.

5.9 Simulation Method

In the previous sections we discussed the scheduling of the matrix elements in order to obtain speedup improvements. This section will introduce the steps needed to simulate the parallel activities by a sequentially working simulation program. The simulating program would repeat the following steps:

1- Choosing the next available processing node PN that will execute the next node ready for processing. A list is kept of all PNs that are allocated for the execution of the algorithm.
2- Choosing the next available node for execution. For each processing node PN to be executed, that is the node with the minimum IPC time (from the list of available nodes), is selected. The selected node may reside anywhere in the queue and not necessarily at the beginning.
3- Fetch operands to complete the node's task. Some of the operands may reside on a remote processor, thus a communication cycle is activated to transfer the operands from the PN where they were generated to the currently selected PN.
4- Execute the node by the selected processor. The action defined by the node's task is executed on the processing node PN.
5- If needed special cycles for the results transfer are activated. Subject to the user requests, there may be a number of options for the transfer of results. Such communication enhancement systems include one-to-all Broadcasting and Caching.

5.10 Simulation Parameters

The basis for a model for parallel computations is to be able to predict the efficiency of execution of a parallel computation on a given architecture, and to study the effect of varying different configuration parameters of the computation. The parameters of interest to this research are as follows:

1- Shared memory / Message passing model:

The interaction between the nodes of a computation can be specified using a message model, a shared memory model, or a mixture of the two. This choice is based on the support by the host architecture and the volume and size of the information being exchanged.

2- Granularity of the *node*:

The efficiency of the computation is directly dependent on the size of each schedulable node of computation. Larger nodes would usually reduce the amount of parallelism possible, while smaller nodes could entail additional overheads of data movement.

3- Computation structure:

This includes the specification of synchronization and sequencing of nodes that compose the computation.

4- Communication Structure:

It is also desirable to provide the processing nodes with special hardware mechanisms to enhance their operation. Such enhancements should effect both the areas of computation and communication capability of the computing architecture. The

computation power of a processing node dealing with numerical algorithms, such as ICCG, will significantly improve by the addition of dedicated vector processing hardware. The communication speed to transfer data among the processing nodes will benefit from a special hardware. Both One-to-All and Caching mechanism introduce great advantages to system.

5- Host architecture:

All the earlier mentioned parameters are implicitly dependent on the choice of the host architecture.

5.11 Conclusion

The CPM is a valuable tool to enhance the operation of parallel programs. We have described in this chapter a method to recognize the critical path of a given network of tasks. Scheduling tasks of a parallel algorithm using the CPM would improve its performance.

The application of CPM to the scheduling of sparse matrix structures will identify parallelism available.

5.12 References

[Berry-85] Berry, Orna and David Jefferson, "Critical path analysis of distributed simulation", Proceedings of the conference on distributed simulation, January 1985, pp. 57-60.

[Carre-79] Carre, B., "Graph and networks", 1979, ISBN 0 19 859 6227,

[Haley-83] Haley, Paul V., "Adding knowledge to the Critical Path Method", Proceedings of the 14th annual Pittsburgh conference on Modeling and Simulation, 1983, pp. 1233-1237.

[Kasahara-84] Kasahara, H. and S. Narita, "Practical scheduling algorithms for efficient parallel processing", *IEEE transactions on Computers*, 1984, Vol. C-33, No 11, pp. 1023-1029.

[Kohler-75] Kohler, W., "A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems", *IEEE transactions on computers*, December 1975, pp. 1235-1238.

[Legrand-89] Legrand, Alain, "A graphical critical path analyser", *Australian electronics engineering*, March 1989, pp. 36-39.

[Shieh-91] Shieh, J. J. and C. A. Parachristou, "Fine grain mapping strategy for multiprocessor systems", *IEE proceedings-E*, Vol. 138, No. 3, May 1991, pp. 109-120.

[Yang-88] Yang, C. and B. Miller, "Critical path analysis for the execution of parallel and distributed programs", Proceedings of the 8th international conference on distributed computing systems, 1988, pp. 366-373.

[Zimmerman-82-1] Zimmerman, S. and L. conard, "Programming the critical path method in Basic", *Byte*, July 1982, pp. 378-390.

[Zimmerman-82-2] Zimmerman, S. and L. conard, "Programming PERT in Basic", *Byte*, May 1982, pp. 465-478.

Chapter 6

The simulation program PARASIM

6.0 The simulation program PARASIM

This chapter presents a **CAD** software package which was developed especially for this research and used in evaluating the performance of a user-defined Parallel Processing Architectures (**PPA**). This package is a general purpose scheduler and simulator of PPA with the added facility of utilizing the Critical Path analysis during the scheduling of tasks.

The Simulation of computer systems is the only method for estimating performance of new designs and new configurations without implementation of the desired system. A simulation method can include factors that are very difficult to incorporate into an analytical model, such as communication, synchronization and system overheads. Also, the workload does not have to be described by probability distributions but the actual algorithms can be made to execute on the simulated computer hardware and software. Simulation has another useful outcome which is to determine possible improvements of existing systems, for evaluation of computer network and as a design tool.

To simulate the parallel machine and its behavior under certain algorithms, we need to simulate the behavior of the processors, memory modules, interconnection networks, algorithm and the interaction between these components. A programming tool that performs both simulation of the parallel machine and the analysis of critical paths for parallel programs has been developed by the author. This tool determines the critical path for the program as scheduled onto a parallel computer with N processing elements.

This software package is based on **Discrete Event Simulation DES** (refer to chapter 5.0), hereafter referred to as **PARASIM.** PARASIM stands for Parallel Simulator. It is implemented entirely using C programming language

(refer to Appendix-B). PARASIM was entirely written in C and was tested running on UNIX, MS-DOS, IBM Risc 6000, HP and SUN workstation. It was also interfaced with the MEGA CAD package to simulate the Incomplete Choleski Conjugate Gradient ICCG solution of matrices which arise from real design models.

6.1 An Overview of PARASIM structure

The program is designed to be flexible in the amount of information it provides. In order to provide detailed outputs for debugging and design verification, it includes a number of output statements. However, these produced far more output data than is needed for most runs. The output data is stored into the log file associated with each run. The output data which is primarily concerned with the overall performance level of the simulated system running the algorithm under test. During the execution of **PARASIM** each option selects the amount of detail in the output data to suit the purpose of the run.

The simulation program PARASIM will accepts input from the user. The input may take one of two forms. The first form is a three file information which contains information about the algorithm which will run on the simulated parallel architecture. The second form is to enter the network structure by the user. PARASIM supports facilities for creating, editing, displaying and printing the network structure and its execution results. PARASIM is equipped with different options to simulate the different connections and communication delay costs.

6.1.2 Simulation Steps

The phases of the PARASIM program operation are as follows:

Phase 1: Preparation of application programs

The steps for preparing the program for running on the simulator are described later.

Phase 2: Menu and dialogue

Data entry using menu-driven user interface

Creation of the network structure.

Phase 3: Critical path analysis

Forward pass execution.

Reverse pass execution.

Identification of the critical path for the network.

Phase 4: Scheduling and Simulation

Multiprocessor simulation.

Statistical data collection and Reporting

Phase 5: Forms of Output

The output of the program may take one of two forms:

(i) Screen Display, or

(ii) Text file output.

ASCII output files

Word processor interface (Latex - like)

Printer support (Postscript)

CAD package interface

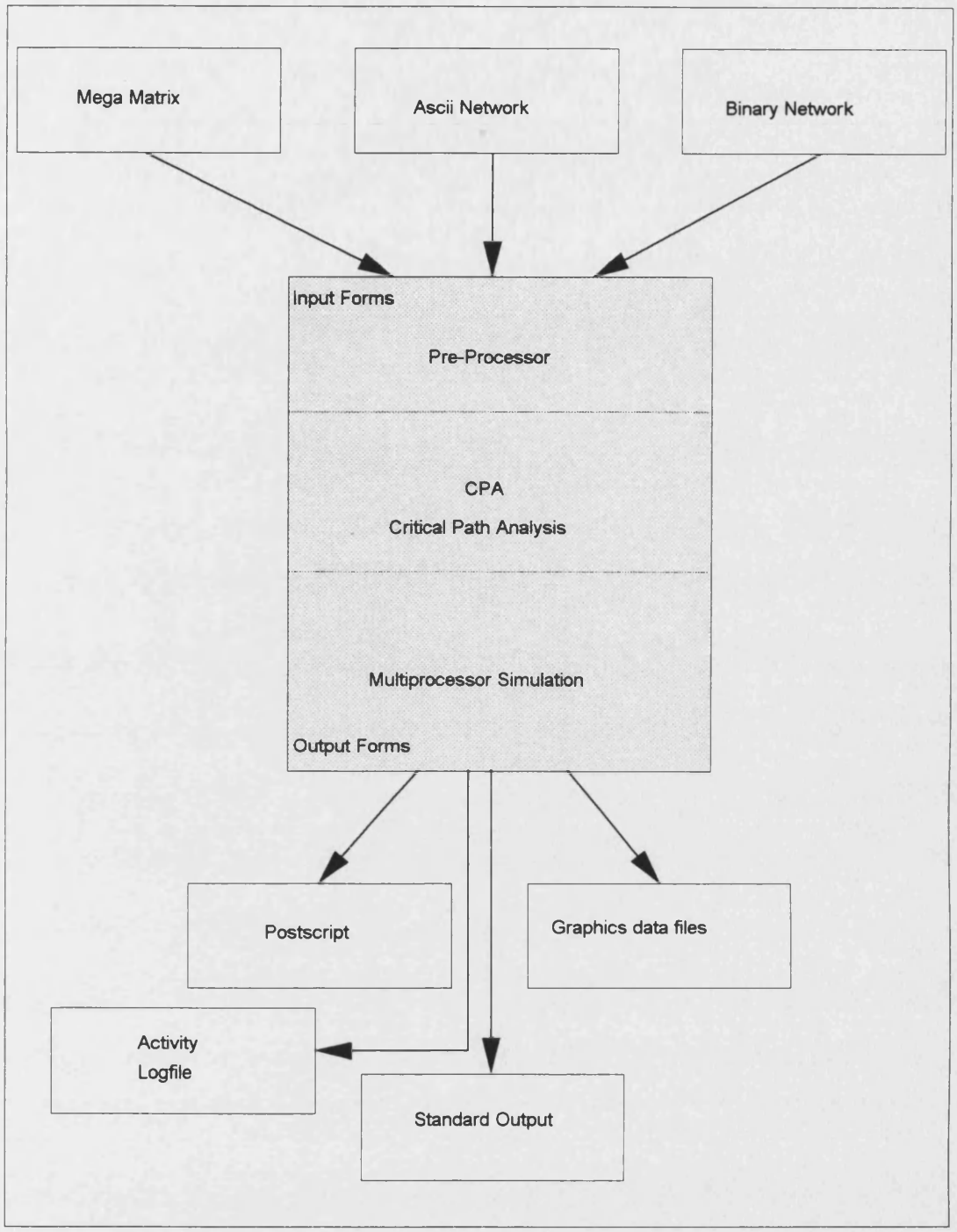


Figure 18: Forms of Input and Output to/from the simulation program PARASIM.

Figure 18 depicts the different forms of input to the **PARASIM** program. Among the forms of input is the "Mega Matrix". This form accepts input from the Mega CAD software. The matrix can also be described using Ascii files. The other form of data input to PARASIM is through "Binary network

description files". These files represent the interconnection of the network to be simulated. The different forms of input are described in this chapter, whereas the forms of output will be discussed under the chapter discussing the results of the simulation.

The following sections will describe in detail the operation of the above phases.

6.1.3 Data Structures of the simulation program

PARASIM creates three data structures to represent the algorithm for the simulation of parallel computing systems. The network contains information on the algorithm structure, parallelism, and costs for execution and communication time. These pieces of information is expected to come from the user. Node, Tasks and Link records will be explained in this section. The Three structures are as follows:

1- Node structure:

Nodes represent an operation being performed on some input data to produce some output(s). The nodes of the network are numbered starting from 1. Figure 19 shows the *Node* structure.

```
typedef struct node {
  struct node *next,
              *queued_next;
  struct link *fwd_link,
              *rev_link;
  struct task *task;
  int time_taken,
      node_id,
      proc_no,
      status;
  int critical;   }
```

Table 3: C code declaration of Node structure.

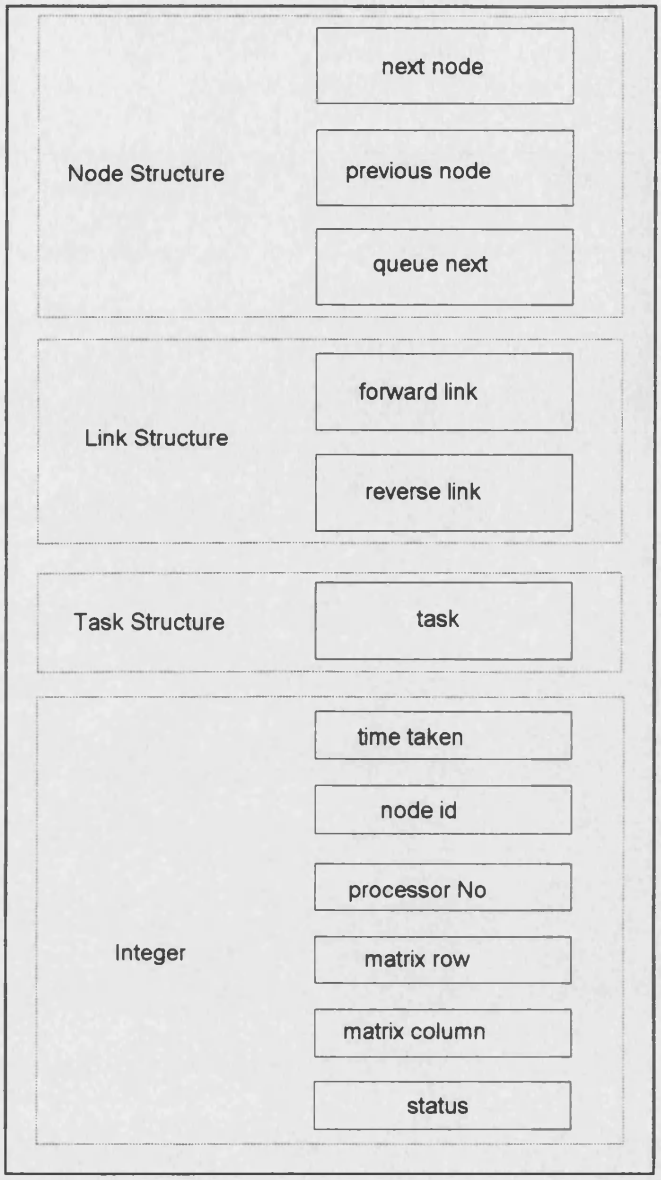


Figure 19: Graphical presentation of Node structure.

The *head_node*, which is created at the start of the simulation, is the pointer to the start of the network representing the problem being simulated. The *head_node* is not itself a part of the dataflow description but it points to the linked list of nodes and is used to access the linked list during the different phases of the simulation. Inputs to the dataflow network from the outside world are presented by links from the *head_node* to the first nodes of the dataflow network. Similarly outputs to the outside world are represented by links to the *head_node*. The following attributes are maintained for each node and are stored in the node structure.

NODE->**id:** To allow the user to identify nodes, each node is assigned a unique identification number when created.

NODE->**fwd_node:** For the purpose of systematic access to all nodes they are connected into a double linked list so that each node record contains a pointer to the next node in the list.

NODE->**fwd_link** & NODE->**rev_link:** The node record contains two pointers to Link structures (i.e. fwd_link and rev_link). The first points to a list of Links where the node is to supply its output. The second is the list of Links which input data to the node. If a node has no Links in a given direction the relevant pointer will be set to Nil.

NODE->**task:** In addition each node is assigned a task which indicates the action to be taken during execution. A pointer to the Task structure is defined.

2- LINK structure:

Links represent data paths between nodes. When data is expected to move from one node to another, a link is established between both nodes, starting from the source node and ending at the destination node. These Links are a list of input operands and another list for the destination of the results. Figure 20 shows the *Link* structure.

```
typedef struct link {
  struct node *fwd_node,
              *rev_node
  struct link *fwd_link,
              *rev_link,
              *next_link;
  int comm_time,
      et,
      lt;
  char input_available;
  char output_available;}
```

Table 4: C code declaration for Link structure.

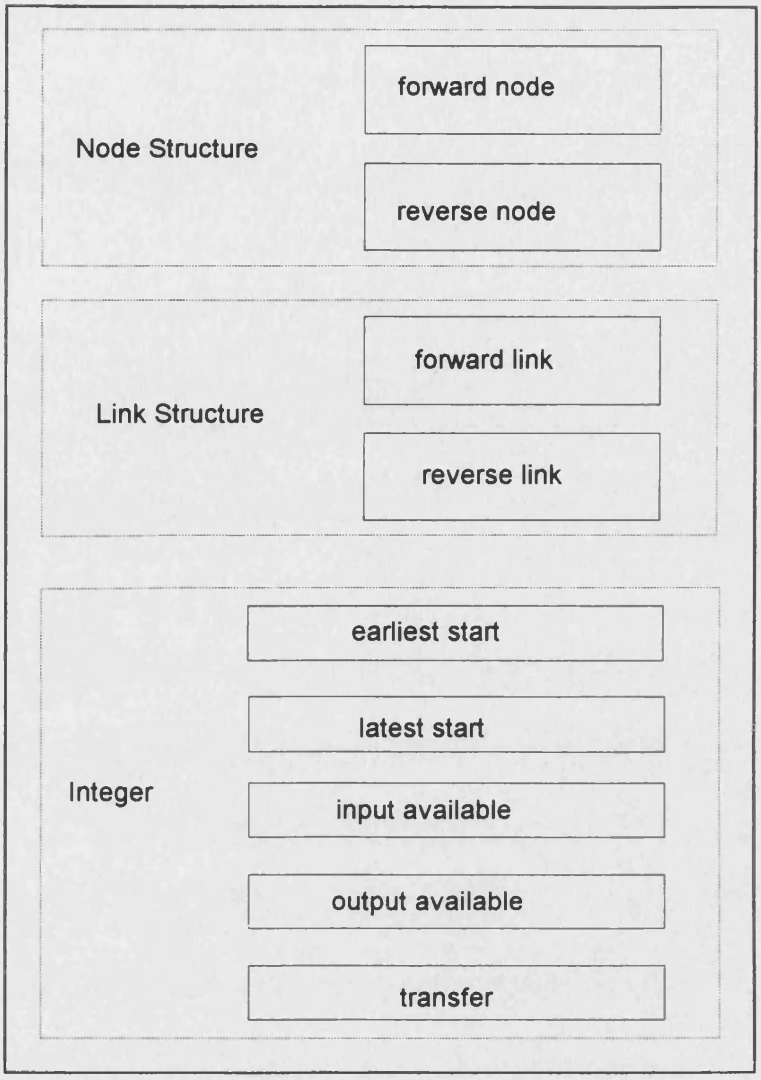


Figure 20: Graphical presentation of LINK structure.

In each Link record there are two link pointers, which represent the data inputs and outputs to node. A linked list connected to the NODE->rev_link is pointing to the list of input operands, whereas the NODE->fwd_link is pointing to a list of destination nodes.

LINK->**rev_node:** This is a pointer to the node where the data comes from.
LINK->**fwd_node:** A pointer to the node to which the data should go.

3- TASK structure:

In order to group the repeated tasks that appear in the network, the Task structure was implemented. Task structure identifies the nature of the task being executed on the node. Each node must have a task associated with it. Tasks can be defined with different arithmetic, move and logical operations and may have an arbitrary task duration. Each task represents an indivisible sequential computation (which may be arbitrarily complex).

Simulation of Looping and Branching:

The user can specify any number of operations, including conditional and loops. However, we should note that although PARASIM allows looping, it still considers it to be a single block of instructions performed by the calling processor node. Figure 21 shows the Task structure.

```
typedef struct task {
   struct task *next;
   char name[20];
   int id,
       time;    }
```

Table 5: C code declaration for Task structure.

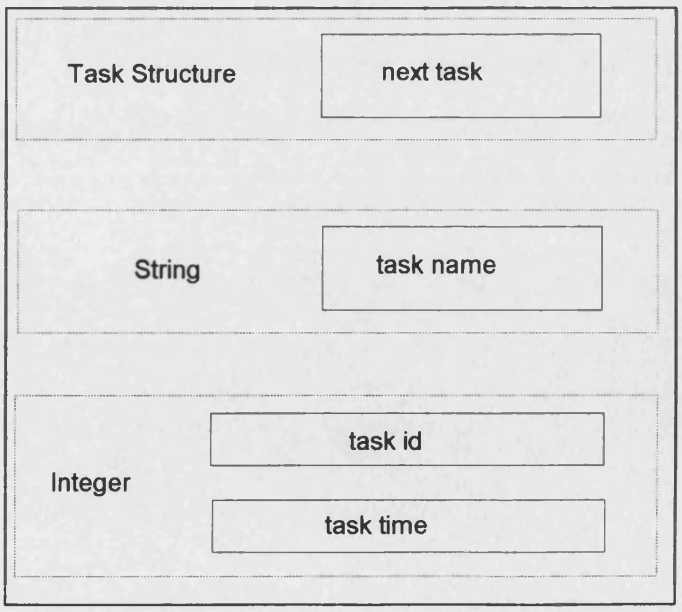


Figure 21: Graphical presentation of TASK structure.

The Task records are linked to each other in a linked list for easy access. The start of this linked list is a record pointed by *head_task* created at the Network generation phase of the program.

TASK->**name:** Each task record contains a name field, which is a string of characters. This field describes the operation to be carried out by the connected Node.

TASK->**id:** A unique identification number is allocated for each Task record. Task numbers start from 1.

TASK->**time:** The time needed by the task to be executed, when the inputs are ready, in time units. Time is an integer variable denoted by *ut* (unit time), in which both the task execution time and the IPC time is measured.

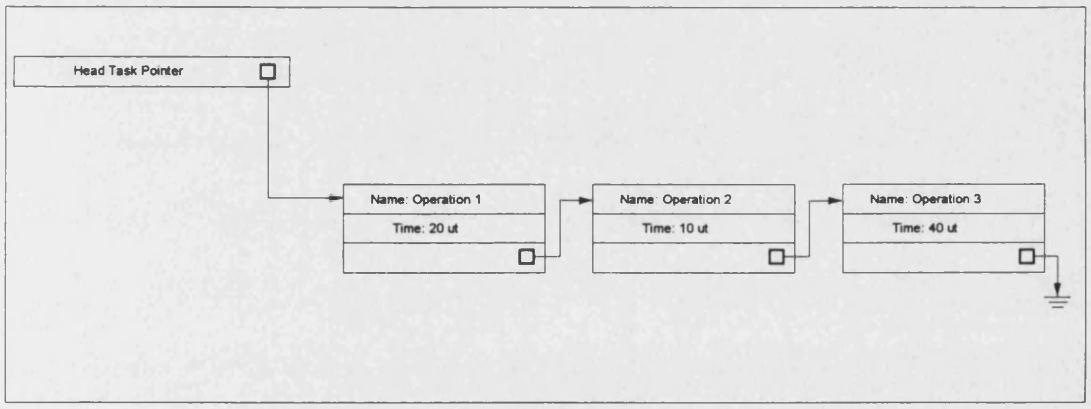


Figure 22: Graphical presentation of the Task list.

Figure 22 shows the structure created inside PARASIM for the organization of tasks. It takes the linked list format. The structure starts with the *head_task* and keeps information about every distinct task in the network. Whenever a new task is encountered, the list is checked and if not suitable task was found then a new structure will added to the list.

Tasks are either a single-operation or a multi-operation structures. But how task time is computed on multi-operation tasks? Once the cost of different single computations have been identified, the cost of executing a multi-operation task can simply be found by adding the time of executing instruction that constitute the task.

6.2 Implementation of PARASIM

PARASIM is not only a multiprocessor simulator but also a scheduler with Critical Path Analysis. We should keep in mind that the PARASIM simulator runs on a sequentially organized uniprocessor computer. Therefore, the actions of the processing nodes and other subsystems defined in a simulated system are not taking place concurrently, but sequentially. PARASIM executes the following phases before it commences the execution of the multiprocessor simulation:

1- Network Creation.

2- Scheduling strategy.

  2.1- Network Forward Pass.

  2.2- Network Reverse Pass.

  2.3- Critical Path identification.

3- Simulation of architecture.

The flowchart in Figure 24 shows the steps of the PARASIM program simulation steps.

6.2.1 Network Creation

The complete network structure that represents the algorithm to be simulated is created using an option from the program's menu. This option, which can be selected from the main menu, creates the network data structures needed by the simulation program. The network is comprised from Nodes, Links and Tasks and is an image of the algorithm as it should be seen

by the simulation program. The whole algorithm steps are created and organized in the memory of the computer.

The algorithm is split into a number of distinct identities, as discussed earlier in this thesis. These identities or Nodes will interact with each other by the exchange of information and data. The size of each of the Nodes depends on the granularity of the splitting strategy. PARASIM offers a number of selections to describe the granularity of the algorithm, although the implemented program only serves the Backsubstitution algorithm but it can be easily modified to cater for other problems. The network representing the algorithm being simulated should be created before any simulation can take place. The connections between the Nodes and Links is governed by the algorithm nature and the granularity of the problem.

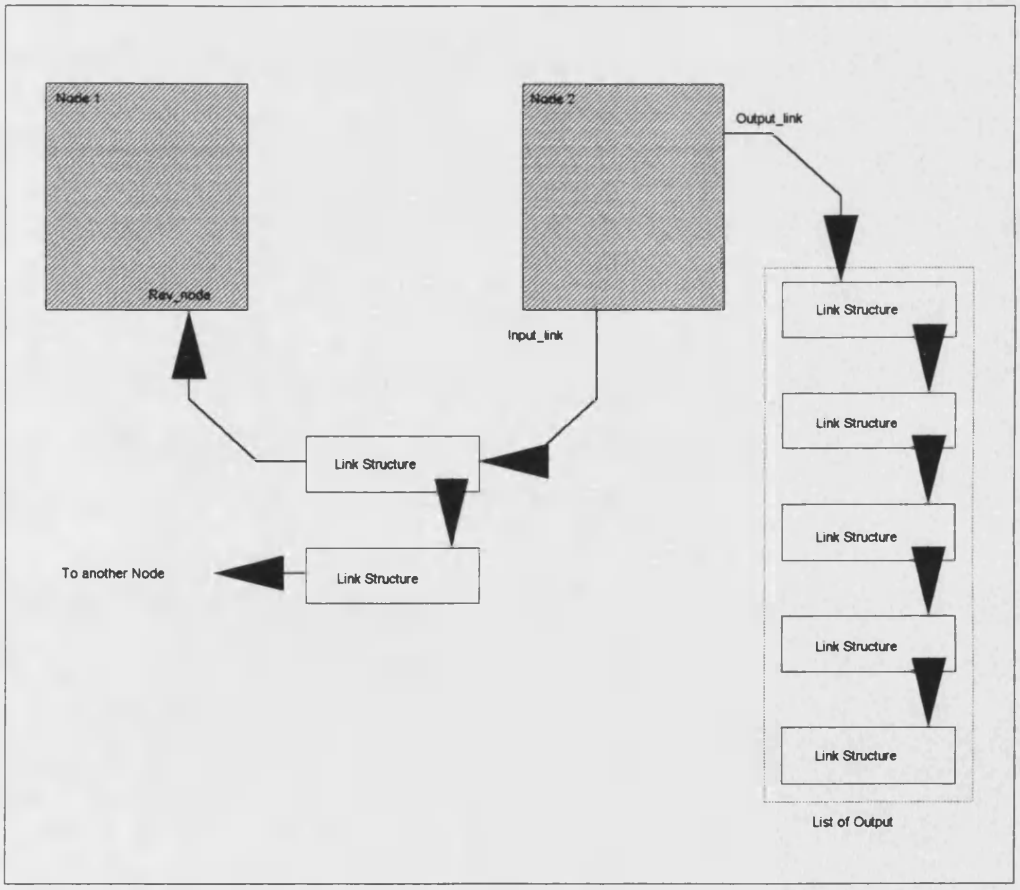


Figure 23: A segment of the network presentation which includes 2 nodes.

Figure 23 shows the internal program presentation for two nodes (namely, $Node_1$ and $Node_2$) linked together. There is one operand that is sourced at $Node_1$ and has to feed $Node_2$. The linked list pointed to by *input_link* represents the operand sources to $Node_2$.

PARASIM offers a number of input forms of data that represent the problem. The program's input data may take one of the following forms, as described in Figure 18:

1- Binary network information.

2- Ascii network information.

3- Mega matrix format.

If the user selects the Binary data input format then the following three data file should be available for each network structure to be simulated. These files are as follows:

1- The Nodes data file:

This file contains the data and information about the Nodes of the network. The total number of Nodes and the task assigned to each Node is an example of such data.

2- The Links data file:

This file contains a list of connected Nodes. Each line in the file has a pair of integers represent the Node identification numbers, the first number denotes the source operand and the second number denotes the destination Node.

3- The Task data file:

This file stores the information about the different Tasks that are executed by the algorithm. The total number of Tasks, Task name and duration is stored in the file.

The only difference between the Binary and Ascii data input format is the method used for saving the data files. The Ascii format allows the user to inspect the contents of the file in a readable format. PARASIM also provides an option to convert a network structure from Mega to Binary definition.

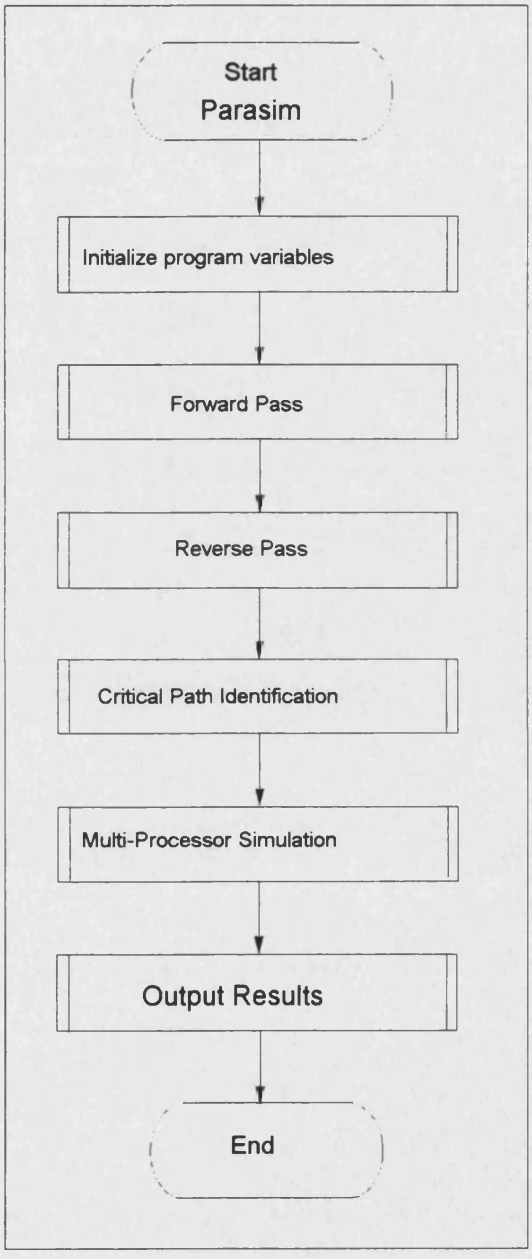


Figure 24: Flowchart of PARASIM steps in operation.

6.2.2 Basic Model Assumptions

FIGURE 24 shows the phases for execution for a node. The model used by **PARASIM** assumes the following phases of operation:

1- Phase 1: The *Communication phase* for executing a node can start whenever the inputs to the node are generated by the processing node.

2- The transfer of data is started by assigning one link to the *active_link* and the data is transferred to the processor executing the node.

3- PARASIM will apply one of the following two cases to obtain the link transfer time:

3.1- Local Transfer Time for transfer of data that is originated from the local memory.

3.2- Remote transfer time for transfers from a remote memory module. The bus contention may affect the transfer I shared memory bus is used in the simulation.

4- Phase 2: The node will begin execution only when all the data is transferred to the processor. PARASIM will spend a number of cycles equivalent to the task cycles linked to the node under execution. The *active_cycles* of the processor will be updated accordingly.

5- Phase 3: As soon as the node is executed the status of the links taking out the data will be marked as *input_data_available.*

6- Phase 4: The node is executed and the output data is ready for transfer to other processors.

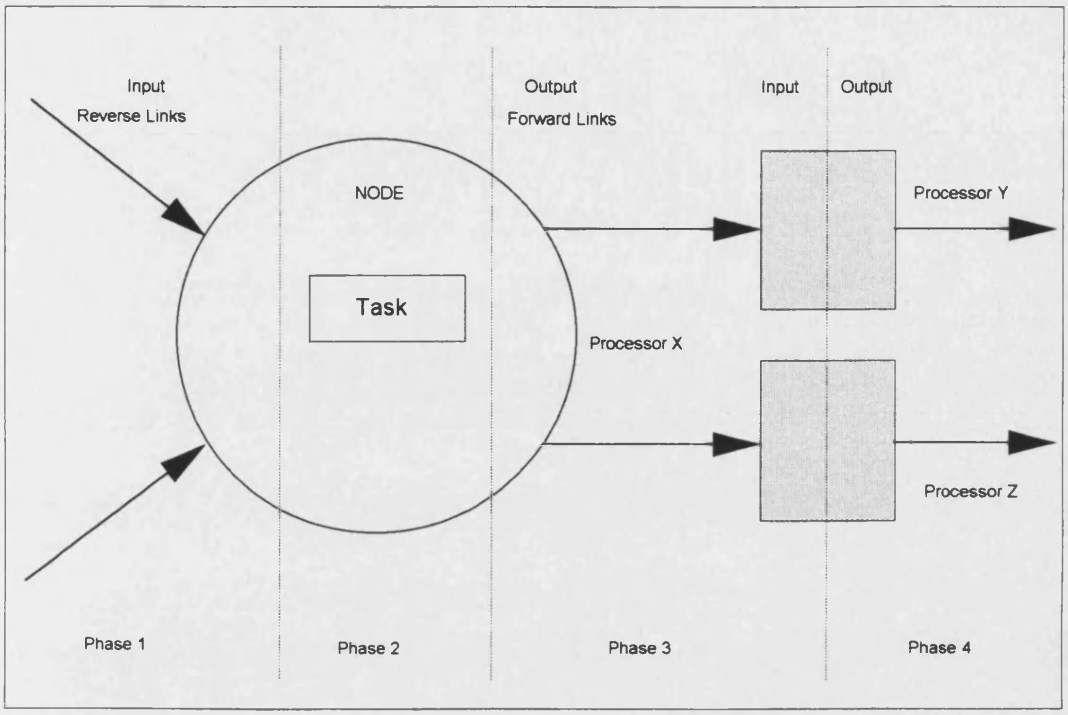


Figure 25: Phases of execution for a NODE with input and output LINKS.

6.2.3 The Window method

Before we proceed any further let us discuss the *Window method* which is devised especially for PARASIM. This method was implemented to improve the execution and response times of PARASIM. In the Window method the execution of one clock cycle does not depend on the size of the network, but only on the number of nodes currently active within the Window. Nodes are considered to be active if they are assigned a processing node to perform the execution. A list is kept for all active nodes. The nodes will join this list when they are ready for execution, and leave the list upon completion and generation of results.

6.3 The Scheduling Technique

6.3.1 PARASIM schedule

PARASIM schedule proceeds as follows:

1- At the start of simulation the clock is set to time = 0. All nodes that have no dependency constraints (i.e., have no predecessor nodes) are ready for execution. These nodes are inserted in the *Window_Queue*. The procedure that performs this task is *Put_All_ready_Nodes*(). These nodes will be distributed among the available processors to commence the simulation of the network.

2- All processors are free at time = 0. They attempt to remove a node from the *Window_Queue* and execute it.

3- If a processor is available and no nodes are in the *Window_Queue*, then the processor will remain idle until a node is entered in the *Window_Queue.*

4- When the processor completes the execution of a node, it will look for the next node in the *Window_Queue. Nodes* that are successors to the completed node and have no other unexecuted predecessors are entered into the *Window_Queue.* A node would be assigned to the requesting processor. This continues until all tasks have been executed.

5- Only nodes designated as Critical will be allocated to processor 0, whereas any other node could be allocated to any other processor.

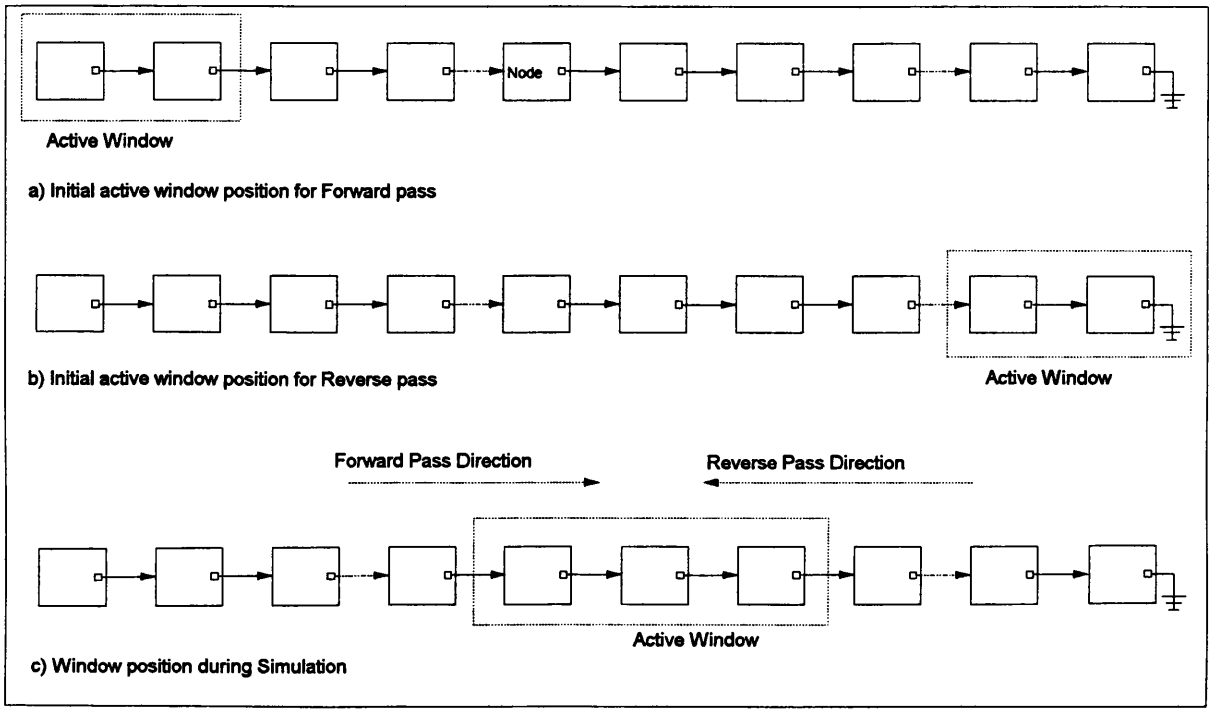


Figure 26: Initial Window position during the simulation steps.

Forward Pass

This procedure aims to identify the *earilest_start* times for all the links within the network. It passes through the dataflow structure once in forward direction. The integer variable *clock* is used to indicate the number of clock pulses elapsed since the start of the pass through the dataflow network. It is reset to zero at the start of the procedure.

The procedure starts to calculate the state of the network dataflow after every clock pulse. This procedure continues until the calculation is completed (i.e., all links were assigned a value) or an infinite loop in the dataflow is detected.

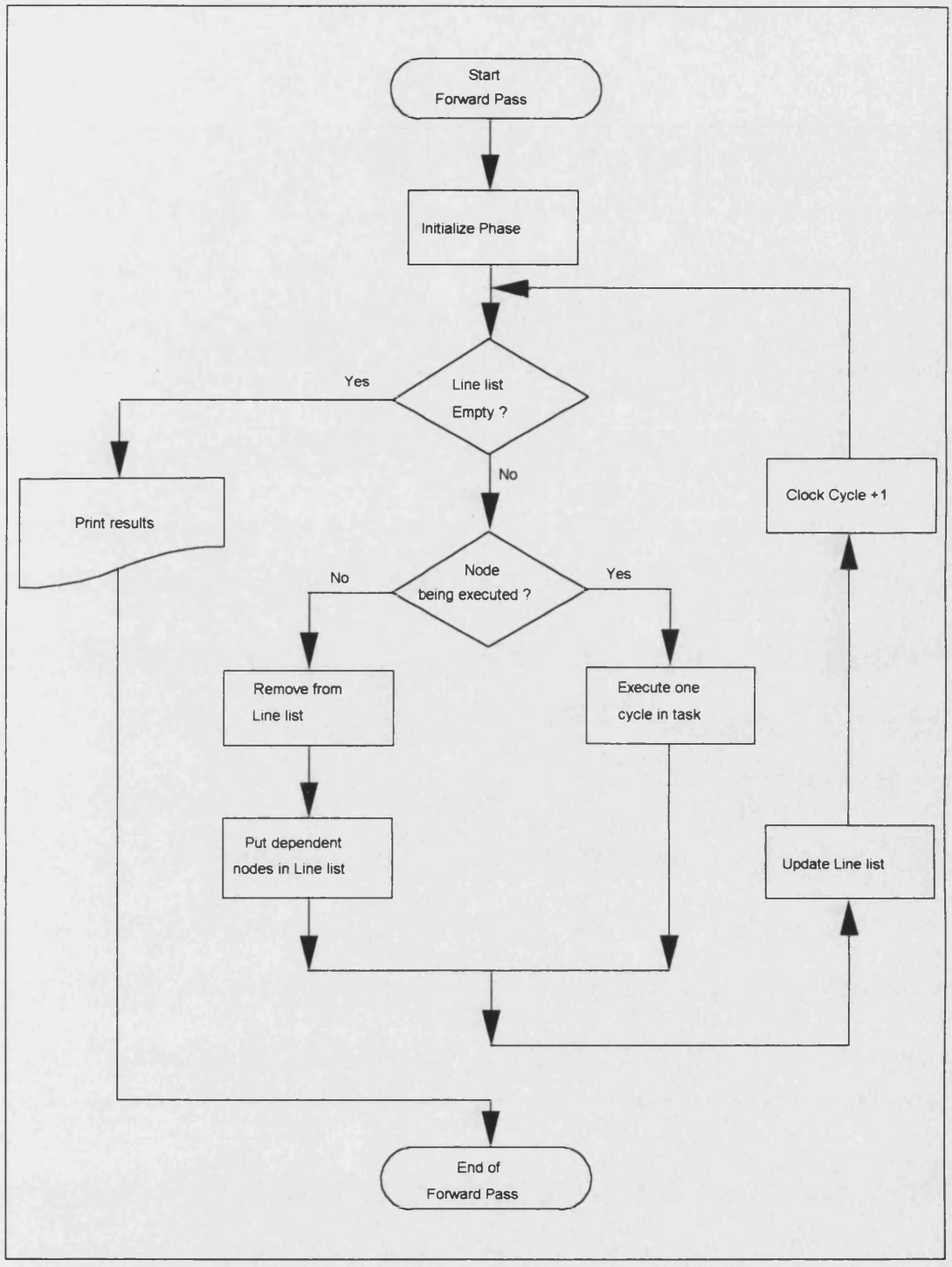


Figure 27: Flowchart of *forward_pass()* routine.

The procedure performs the following steps:

1- The node linked list is read through. Every node is checked to see if it is ready for execution by reading through its linked list of input links, if all

these have their *output_available* parameters set to true the state of the nodes, execution is examined.

2- If its *time_taken* parameter is less than or equal to the time of the task record associated with the node the *time_taken* parameter is incremented. It is necessary to increment the *time_taken* value when it is the same as the task time parameter to indicate that the nodes execution has been completed.

3- If its *time_taken* parameter is the same as the task records time parameter it indicates that the task has just been completed.

4- The output linked list of the node is read through setting all the *input_available* parameters to true. This indicates to the following nodes that this data is now available.

5- The node linked list is read through again to adjust the state of various links. Each node forward link is read through. If the *output_available* parameter of a link is still false but its *input_available* parameter is true this indicates that the data has just become available and hence the links *earilest_start* parameter is set to the current clock value, and the *output_available* parameter is set to true.

Reverse Pass

This procedure aims in identifying the *latest_start* times for all the links within the network. At the start of this procedure the variable clock is set at the optimum time for the computation and each node records *time_taken* is set at one plus the time to complete to complete its given task. As this procedure works backwards through the dataflow the first step is to mark all the *output_available* parameters of the links to the head node as false. Thus, the procedure starts when the computation is in fact complete and works back in time to the original position.

As with the *forward_pass()* the procedure consists of a routine being repeated a number of times, each repetition of this routine results in the clock

variable being reduced by 1 and the end of the pass is indicated by this variable reaching a value of zero.

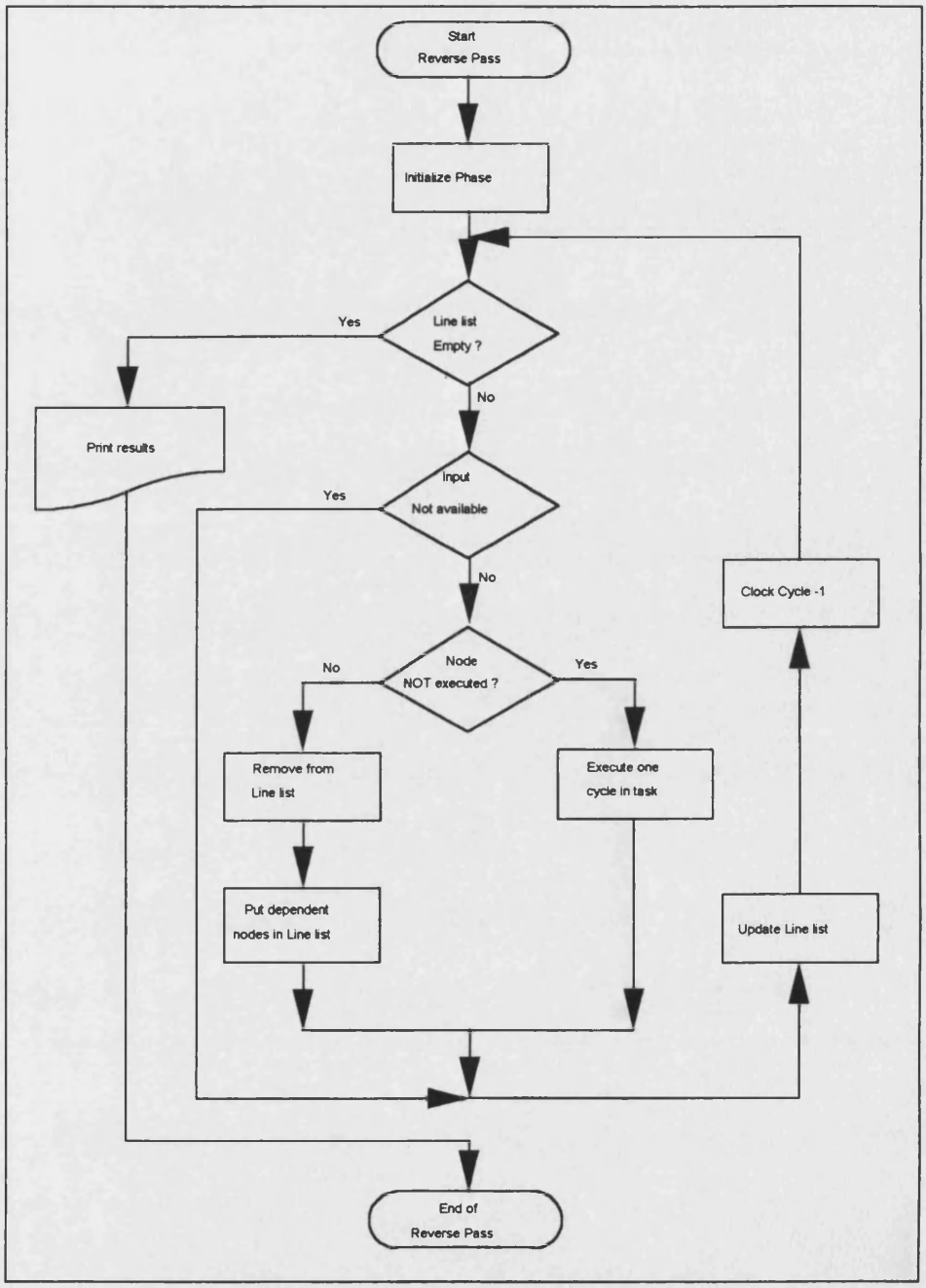


Figure 28: Flowchart of *reverse_pass()* routine.

Critical Path Computation

During the execution of the *reverse_pass()* routine each communication link will be assigned its *latest_time*. An extra comparison is added to the code to check the following condition:

earliest_start = latest_start

If this condition is true the node associated with this link will be considered a CRITICAL node (i.e. it resides on the critical path). A counter of the number of critical nodes is also kept. Another parameter computed during the *reverse_pass()* is the total execution time of the critical path. This represents the time spent for the computation of the critical path tasks. It is also used to find the best speedup achievable for the given network.

After the execution of the *forward_pass()* and *reverse_pass()* routines the critical path is identified and the best time possible for the execution of the network can be evaluated. It is now possible to obtain results based on finite number of processors and a non-zero interprocessor communication time.

6.4 Multiprocessor Simulation

PARASIM provides the user with the ability to select the type of architecture to be used in the simulation. The program offers a number of architectures that can be simulated. These architectures are:

1- Distributed memory system,

2- Shared memory system, and

3- Additional communication enhancement hardware features.

Combining both the capability of simulation different forms of parallel computing architectures, together with the facility to allocate nodes using the critical path method, produces a powerful scheduling and simulation tool.

The procedure *proc_sim()*

This procedure is used to obtain execution time results for the execution of the network on a number of processors. *proc_sim()* is called first for one processor, then the processor count is increased and called again. This is repeated until either the best achievable time is achieved or the maximum number of processors is reached.

```
Repeat:
    Select a ready node for execution.
    Select a processor to run the node's task.
    Assign the node to the processor.
    Processor will execute the node's task.
Until all nodes are executed.
```

Table 6: The Basic scheduling algorithm steps.

A round-robin selection criterion is used in selecting the next available processing node for execution.

The next section will identify each component used in the simulation, and explains how it is implemented. The main components are as follows:

1- Target machine.

2- Processing Node.

3- The multiprocessor bus.

4- Memory modules.

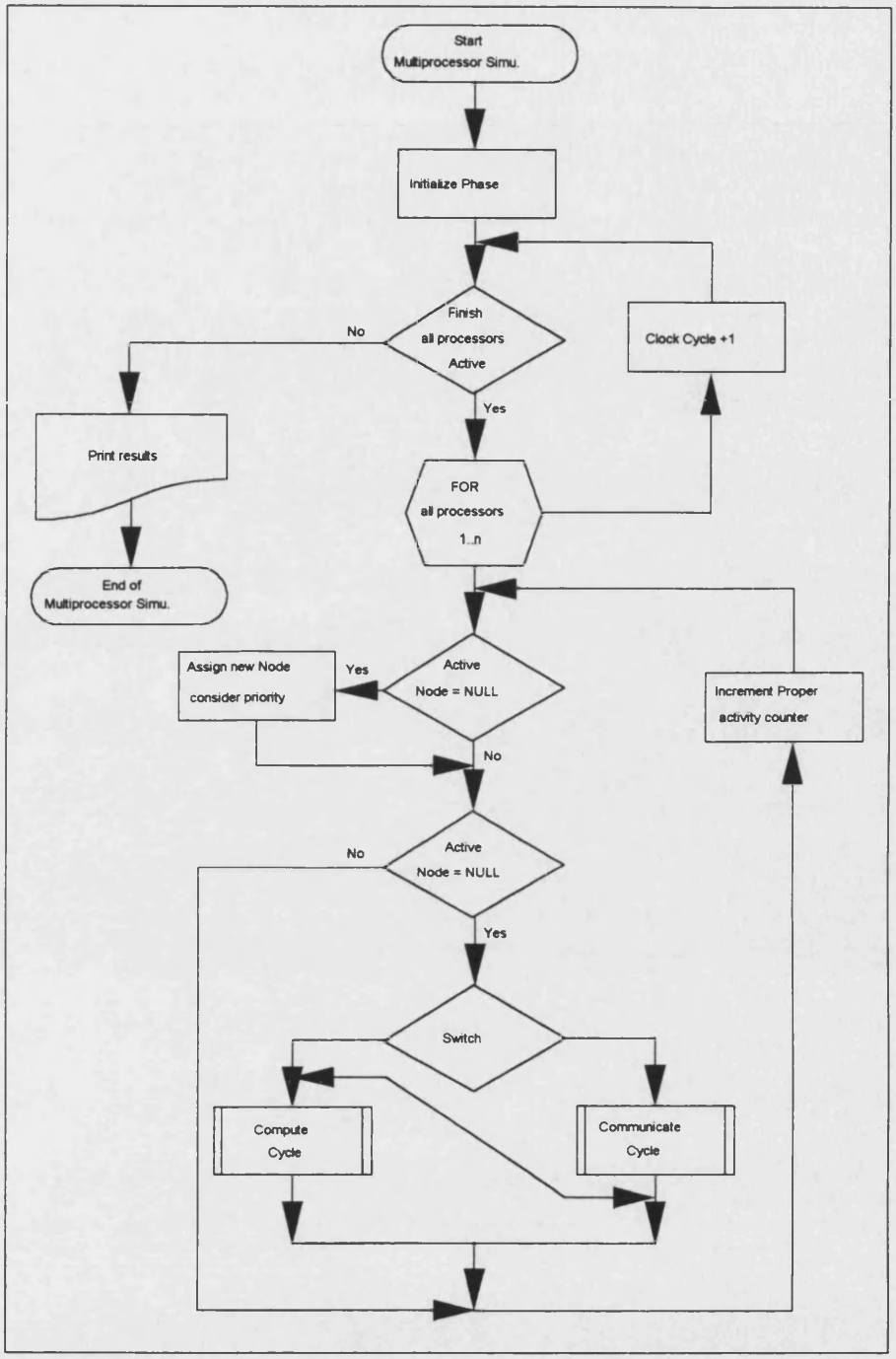


Figure 29: Flowchart of Multiprocessor simulation routine.

The architecture:

The structure of the multiprocessor system to be simulated and its interconnections and hardware features. The computational and communicational model of the architecture to be simulated must be defined before the start of the simulation run. The computational model specifies the

time needed for each computational task. The communication model identifies the structure and cost for all the interprocessor communication activities.

6.4.1 Simulated Architectures Models

PARASIM is capable of simulating both shared and distributed memory multiprocessor architectures. It assumes that the shared memory system is a single time-shared bus system, with all processing nodes communicating through a single bus. The distributed architecture has a fully interconnected network that allows each processing node to communicate with other units within the same system. This is achieved by simulating a bus interconnection mechanism.

Target Machine:

The target machine is assumed to have a general **MIMD** architecture composed of a number of PN. Beside containing local memory to hold code and data, each PN is assumed to incorporate a private communication mechanism.

The parallel machine being modeled consists of a number of processor and memory-module pairs connected by a network. Each memory module consists of a number of memory banks. Each processor and memory module is connected to a pair of input and output ports of the network. Each processor can access its memory module directly and other memory modules through the interconnecting network.

PARASIM can be configured as a uniprocessor or multiprocessor systems. The type of machine to be simulated is specified by the user as the number of processors executing the given network structure. If the user specifies only one processor then the simulation is configured as a

uniprocessor; whereas, specifying one more processor will invoke the multiprocessor actions with the simulation.

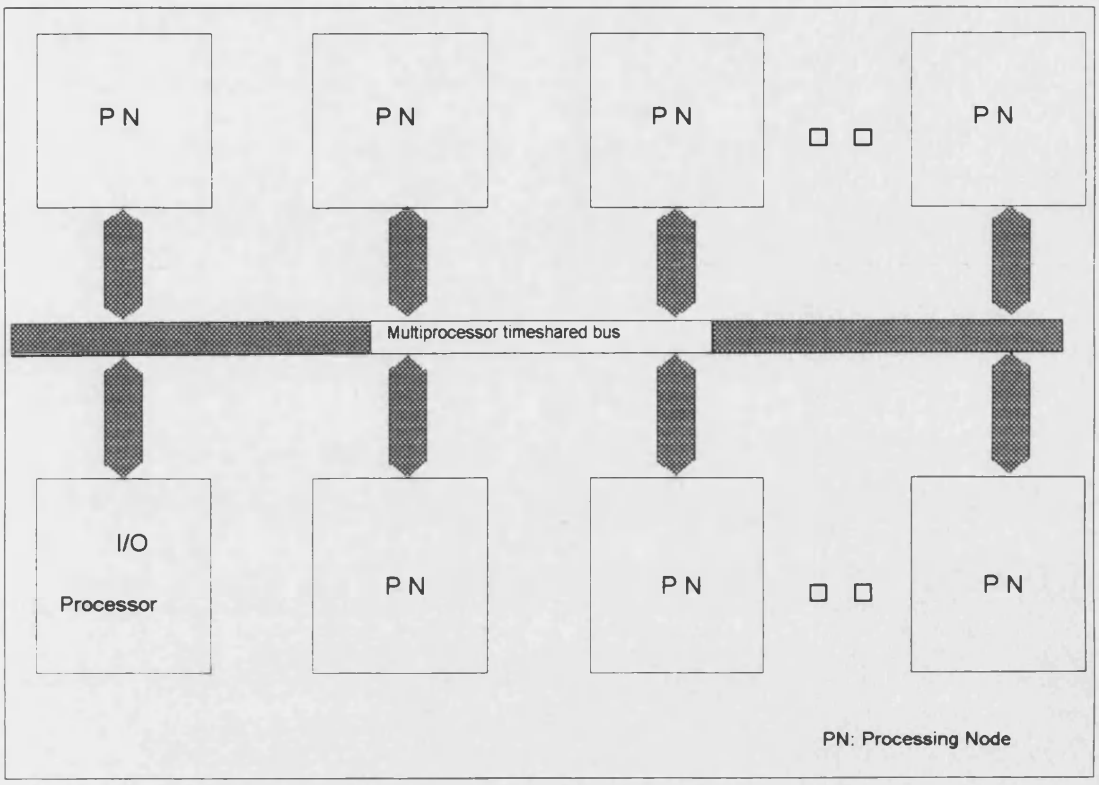


Figure 30: A typical shared memory multiprocessor computer with time-shared bus structure.

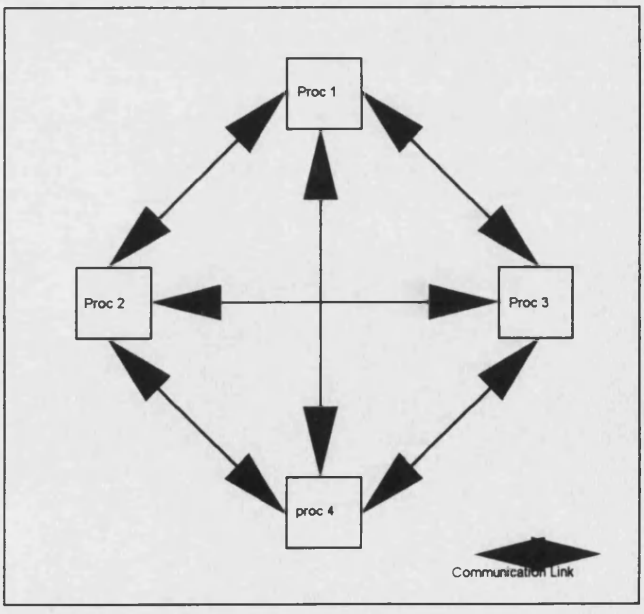


Figure 31: A typical distributed memory multiprocessor computer with direct communication links.

The Processing Node:

PN is intended to form the building block for a parallel multiprocessor system. Each systems consists of a quantity of *n* modules and *m* shareable memory areas (m0). Each processing node PN in the system is addressable and holds unique characteristics and it is an independent computer system with full memory, control and arithmetic capability. It is also equipped with a pipeline feature.

TABLE 7 shows the declaration of the structure that holds information regarding the Processing Node.

```
typedef struct t_processor {
   struct node *active_node;
   struct link *active_link;
   int active_cycles,
       idle_cycles,
       active_comm_cycles,
       idle_comm_cycles,
       active_local_cycle;  }
```

Table 7: C code for declaration the Processing Node structure.

Parameter	Function
active_node	Points to the node being currently executed. Null indicates that the Processing Node is not assigned any node, thus it is free.
active_link	Points to the communication link being transferred. This link is transfers data to the node. Null indicates that the processing node has no communications left to perform.
active_cycles	Time units spent in active operation.
idle_cycles	Time units spent waiting for data communication to take place.
active_comm_cycles	Time units spent in active communication and transfer of data.
idle_comm_cycles	Time units
active_local_cycle	Time units

Table 8: Details of the Processing Node declaration variables.

Memory access model

PARASIM assumes two types of memory systems available in the simulated system. The *Local* memory is the memory location that resides inside the processing node addressing space, whereas the *Remote* memory location resides on another memory module and is accessible by the processing node accessing its information.

In order to model memory access within a multiprocessor system, two parameters are introduced viz., t_{Local} and t_{IPC}. The first of these two parameters is to model the time needed for a local memory location to be accessed by its processor. The second parameter would model the time needed to acquire data from a memory bank which is not local to the processor but it will be accessed through the timeshared interprocessor bus. This time includes arbitration time depending on the activity of the bus.

Multiprocessor Bus Simulations

Modeling the overhead due to bus contention is a difficult problem. Since actual measurements are not possible until the multiprocessor system and the algorithm have actually been designed and implemented, one has to rely on the principles of operation of such structures. To model the effects of bus contention on the parallel execution of the network, our simulator uses a queue model for the bus structure. Every data transfer has to take a finite time and only one transfer takes place at any given moment in time.

The time that a processing node has to wait for a ready data to be transferred from remote memory to its local memory is considered as idle time.

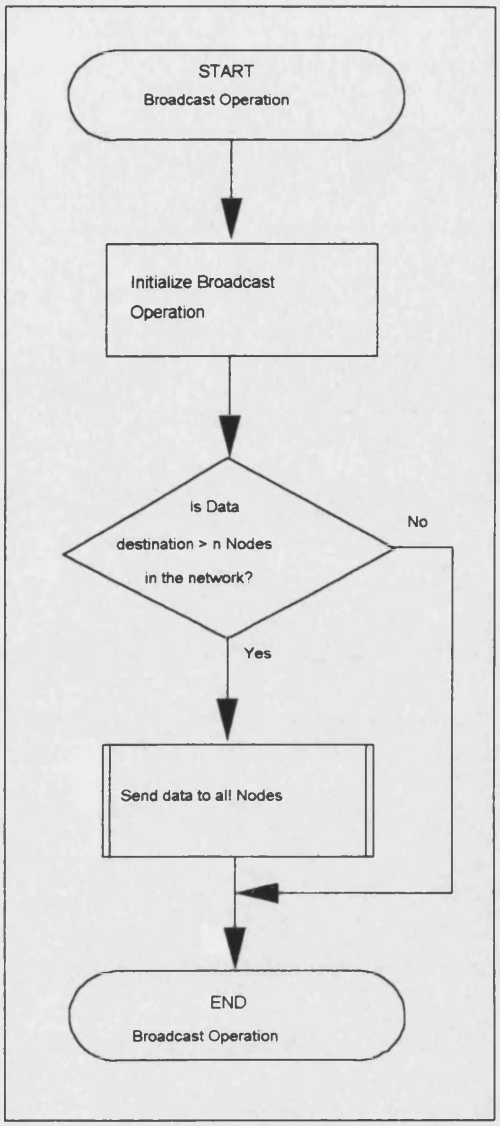


Figure 32: Flowchart of Broadcast operation criterion.

The program also allows the simulation of irregular communication pattern. Data transfer time between any two processors within the multiprocessor may not be always the same, thus **PARASIM** provides a facility to assign different communication time for any pair of processing nodes.

Fetching of remote operands

When a processing node accesses a shared value, a connection is immediately established between the processing node and the processor

holding the referenced value. The data is transfer if the bus is free, else it has to wait until it becomes available for transfer.

The communication logic on board the processing node can issue only one memory request at a time. It cannot make another request if its current request has not been granted.

Special Bus cycle:

We have modeled the bus structure inside PARASIM to possesses a special feature to simulate the One-to-All broadcast mechanism. In this mechanism, the processing node will be capable to send a the results of a node to all the nodes on the bus. Each processing node will be equipped with two types of bus hardware interface units. These units are:

1- Send data unit. 2- Check and Receive data unit.

The above two units will enable the processing node to perform One-to-All communications.

Simulation the operation One-to-All Broadcast:

The One-to-All broadcast operation is simulated in PARASIM by using the following strategy:

1- Nodes that are allowed to broadcast their results will have a special flag associated with them.

2- When the results of these nodes become ready and accessed by any other processing node a multiprocessor bus cycle is initiated. This cycle will transfer the same information not to one processing node only but to all connected to the bus.

The result from the node will only be sent out when a request for this data is received by the communication logic on-board the processing unit. if however the result is used only by the same processing node then there is no need to broadcast to other nodes.

The bus structure will count the One-to-All broadcast as only one data transfer. This improves the performance of the program. If the result form a node has to feed a number of others nodes, then utilizing the One-to-All broadcast will result in a reduction in the traffic on the bus. Only one transfer operation is needed on the bus, but all processing nodes will have a copy of that data locally. When this data is needed by the same processing node for the execution of a different node the data will collected locally, thus avoiding a multiprocessor read cycle.

The One-to-All broadcast mechanism enhances the speedup results of the system by reducing the amount of traffic on the bus between the processing nodes.

Optimized One-to-All Broadcast:

During the experiments utilizing PARASIM as a scheduler/simulator we have found that the in special cases the One-to-All broadcast mechanism does not improve the speedup values. This was discovered to the fact that some overheads are added to the performance of the system. We have devised a condition to benefit from the One-to-All broadcast mechanism. The condition is to allow the node the mark its output for broadcast only if a certain condition is satisfied. This condition is based on the number of output links leaving the node. If the node had output links more than a threshold value, then the node is allowed to broadcast. The links leaving the node indicate a data movement from that node to other nodes on the network. Consider the following case: A node has 10 output links, indicating that the data from this node has to move to other 10 nodes in the network. Let us also assume that the network will run on 4 processing node architecture. There is always some chance that the 11 node system will not run on the same processing node. Thus if we set the criteria to enable broadcast to be 4, we will ensure that each processing node will hold a copy of the result in its local memory.

The Memory System:

The basic clock unit in our model is a machine cycle. It is assumed that the processing node can execute any task in one or more machine cycles. Similarly a memory module responds to a memory request in one or more machine cycles.

Hardware description file:

This is a data file supplied to PARASIM program depicting the different hardware parameters of the multiprocessor system:

1- IPC time and interconnection matrix.

2- Local transfer time (t_{local}).

3- maximum number of processors to be used in the architecure.

Network Description file:

A typical file describing the network structure might take this form:

```
separator - Group 1----------
"Network name"
"network description 1 line"
separator - Group 2----------
number of tasks
task_id1    utime1              "task name 1"
task_id2    utime2              "task name 2"
separator - Group 3----------
number of nodes
node_id1    task_id1    processor    "node name 1"
node_id2    task_id2    processor    "node name 2"
separator - Group 4----------
from_node1 to_node2
from_node2 to_node1
separator ----------
```

Table 9: Contents of Network Description File.

Notes:

Group 1 gives descriptive information about the network.

Group 2 is the network Task definition.
Group 3 is the Node definitions with associated tasks.
Group 4 Network interconnection relationships.
The simulation program is able to generate a data file for the user specifications.

6.4.2 Inter-Processor Communication Models

Introduction

We will model the interprocessor communication structure of a multiprocessor computing system in such a way to reflect the time delay introduced to the system. Processing nodes communicate with each other by sending and receiving messages. The size and direction of the data is defined by the *link* structure of each node. Each processing node knows where and when to send and/or receive data. Communication in PARASIM is synchronized by message-passing.

In order to make the multiprocessor simulation produce more accurate results, the Inter-Processor Communication IPC was considered. This addition would simulate the action taken by the processing node to acquire the operands for its operation.

To introduce communication delays due to data transfers caused by sequential nodes not being executed on the same processor the boolean parameter *comm_time* is included in the link record structure. This is initially set to zero in the procedure *reset_variables()*. It is set to true when any necessary delay has occurred during the second pass of *proc_sim()*.

6.4.3 Operation of multiprocessor simulation routine

This routine is a more complex version of the *forward_pass*() routine. The node record structure linked list is read through and every node that has

not yet been queued (indicated by the queued boolean parameter of the node record structure) has its input link linked list read through to check if all its inputs are available. If they are (indicated by all the link records *output_available* parameter being true) then the node is queued to either *critical* and *non_critical* queue, depending on the state of its critical boolean parameter. Its queued boolean parameter is set true.

1- When the node becomes ready for execution it is placed at the end of execution request queue.
2- The nodes are allocated for execution according to the order of the arrival (First-Come-First-Serve). The next node to be executed is the first node in the queue.

Node allocation or Processor Assignment

The *Window_Queue* will contain a queue of nodes that are ready for execution. These nodes have all their input operands available (i.e., value of the operand is known at the time). Nodes may be selected using one the two methods:

1- Automatic Assignment:

This allocation strategy is used to minimize the communication costs for the data transfer. The processor will be allocated the node which needs the least communication time to acquire its operands. Once the node has been executed, the next node will be selected from a the list of nodes where the output results are going, thus reducing the communication cost for its execution.

2- Arbitrary Assignment:

When the processor completes the execution of a node's task and does not find a suitable node for execution, it will be assigned an arbitrary

node from the *Window_Queue*. This ensures that the processing nodes are always busy, executing the assigned tasks.

6.5 Row output of Simulation

A **PARASIM** simulation run yields two sets of results: The algorithm results and performance results. One can examine the algorithm results by accessing an option in the program's menu, also PARASIM provides options to print and display the network details. The performance statistics can either be displayed on screen or outputted to an Ascii output file containing information on how the simulated algorithm behaved during the simulation execution.

PARASIM is capable of generating a number of data measurements. The resulting data is stored in sequential files for later analysis. Examples of these are:

1. Number of nodes executed.
2. Frequency of each type of node.
3. Data movement.

Speedup:

The speedup *s* attained using n processors to solve a problem is defined by the formula:

$$s = \frac{T_{Seq}}{T_{Par}}$$

The following table shows a summary of the output information generated by PARASIM during the simulation:

Abr.	Meaning
T_{Seq}	Sequential execution time. It is the time needed by a uniprocessor to complete the execution of the job.
T_{PAR}	Parallel execution time
S	Ideal speedup is the ratio of T_{Seq}/T_{PAR}, this assumes no communication time.
n	Number of Processors
T_{Busy}	Total busy processor time (in time units *ut*)
T_{AComp}	Time spent in active computations (in time units *ut*)
T_{AComm}	Time spent in active communication either Local or remote (in time units *ut*)
T_{Useful}	Total useful processor time over all *n* processors useful time is busy time without idle time (in time units *ut*)

Table 10: Output information of the simulation program PARASIM.

Detailed simulation output of PARASIM will be presented in the next chapter. Graphical and tabular results of processor allocation and utilization are being produced.

Applications

Utilizing the simulation program the user can find out the utilization of the PNs. The state of each of the operand transfer actions and the delay generated by other different system actions. The performance of the whole architecture can be investigated and the bottle-necks recognized in an early phase of the architecture design process. To find an optional balanced configuration, architectural elements can be tuned by varying the different system and communication overheads and the delay times of the messages.

6.6 Development phases & Stages

During the development of **PARASIM** a number of stages were implemented. These stages are described below. Writing of the simulator has passed through a number of phases during its development. The time taken to execute a complete run of the program, which included both the scheduling

and simulation of the investigated algorithm. The change from one phase to the next is to improve the overall simulation time of the system being simulated.

Phase 1 All Nodes visited

Go through the complete network and check the status of each node. This method consumes at lot of time scanning nodes that do not need any attention.

Initially, the program would scan **all** the nodes of the network. Starting from the *head_node* and stopping at the *last_node* of the network. This is done at every change of the clock cycle. During this pass the algorithm will check the status of each node, and take the action accordingly. The node may take one of the following status:

1- The node is waiting for at least one input operand to become ready.
2- The node has all its input operands ready. So its task can be executed on an active processor.
3- The node's task is being executed on a given processor.
4- The output result of the node is being produced.
5- The node becomes *dead* and no longer needs any attention. Thus it is removed from the *window queue*.

Also checked during this phase is the status of each processing node. The processing node may have one of the following status:

1- Free processor with no load attached to it for execution. Denoted by *Processor.active.node = NULL.* This processor has no node assigned, and it is ready to execute any other task.
2- A node has been allocated to the processor, thus the processor is no longer free.
3- The processor is fetching the operands of the assigned task from local or remote memory locations.
4- If all operands are ready and reside within the local memory, then the task execution can start.

Phase 2 Window method

To keep a list of nodes that need attention. Nodes that are ready for execution and waiting for a free processor are placed in a queue. The main simulation cycle will check only the nodes that need servicing. Nodes will enter this queue only once.

In order to improve the performance of the simulation, a new method was advised by the author. This method keeps a dynamic list of nodes that need attention. In this context attention means that the node which has ready operands will reside on this list. This node can be executed on any available processing node. This list of nodes is kept by the program, and contains pointers to all nodes that are ready for execution. Nodes are queued into this list if all operands are available, and it is dequeued from the list if the node is assigned to a processing node for execution.

There was only one global clock cycle to synchronize the operation of all processing nodes. All processors during the execution are at the same time slot, (e.g., all processors are at clock say 251).

Phase 3 Distributed Clock

In this phase the program implementation included a distributed clock structure instead of a single clock variable to control all the activities of the simulation. All components of the system that are a function time, a new variable was attached to them. This variable will measure the time for the specified component. No action can take place without the prior checking of the local component clock. For distributed memory systems the processor activity is considered, whereas for shared memory systems the multiprocessor bus activity is considered.

6.9 Conclusion

In this chapter we have presented the software program PARASIM with all its functions and components. It was designed to both schedule and simulate the operation of a distributed computing systems, and to produce performance measurements for any given algorithm. We will use PARASIM to simulate the execution of the backsubstitution algorithm.

6.10 References

[Homewood-90] Homewood, Michael, "Simulation of a Parallel Computing System", Final year project report, University of Bath, School of Electrical Engineering, 1990.

[Sarkar-89] Sarkar, Vivek, "Partitioning and Scheduling Parallel Programs for Multiprocessors", 1989, Pitman Publishing, ISBN 0-273-08802-5.

Part FOUR

Results and Conclusions

This part of the thesis presents the simulation results, discusses the findings and draws conclusions.

Chapter 7

7.0 Simulation Results and Analysis

This chapter presents simulation results for the parallel execution of the backsubstitution algorithm. This algorithm was identified earlier (chapter 2) in this research as the potential element for performance improvement of the Incomplete Choleski Conjugate Gradient ICCG. It represents 70-80% of the computational load for one iteration of the ICCG.

In this chapter we are going to illustrate the different parameters which we have investigated during the course of this research. The models used in the simulation are presented first, Next results are presented for the execution of the algorithm on uniprocessor systems. A number of parameters, affecting both the network structure and the architecture, were changed and the effect of each one is analyzed and discussed. The software scheduler/simulator PARASIM was used as the simulation vehicle to schedule and simulate the algorithm and obtain performance results.

7.1 Introduction:

The simulation results presented in this chapter summarizes our investigations into the performance of the ICCG on parallel computing architectures. The detailed tables of speedup measurements are included later in this thesis in Appendix-A. In summarizing the resultant data the emphasis was placed in three areas. Firstly, measuring internal parallelism. Secondly, identifying how well each separate structured scheme performed and why. Thirdly, assessing the communication overheads in the data transfer.

To measure the internal parallelism of the solution steps of the backsubstitution algorithm, we have converted the matrix structure into a network. This network will represent all arithmetic operations and

communications links required to complete the algorithm steps. We have also identified the critical path within the network. This path identifies the least time needed to complete the execution of the network.

We will investigate a number of parameters that affect the parallel execution of the algorithm, namely, parameters affecting the network structure such as the granularity of the network, and parameters affecting the architecture to be simulated.

7.2 The Data Models used in simulation

We have used a number of problem models in our simulation. These seven matrix problem models, which have risen from a Finite Element Model, differ in both size and sparsity structures. Our interest is not the values of each matrix element but the structure of matrix. From the matrix structure and distribution of elements we will investigate the parallel performance. From these problem models, we have produced 50 speedup result tables, as shown in Appendix-A. In Figure 33 the relationship between the these components is shown.

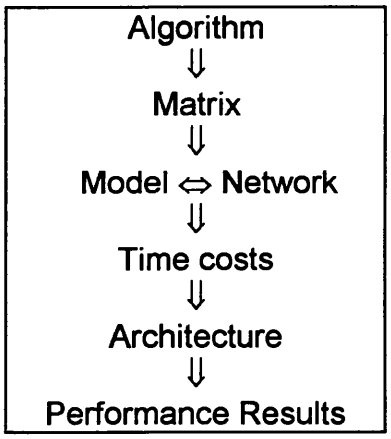


Figure 33: Relationship between Algorithm, Model, Network and simulation.

Table 11 Lists information regarding the model numbers associated with each problem model, and identifies the sparsity percentage of each problem

model. The Model numbers are unique for each model used and *non zero elements* refer to the number of elements in the upper triangle matrix. The table also shows the percentage of non zero elements in the matrix, known as the sparsity ratio. In the problem models used this sparsity ratio is between **0.29% to 1.52%.**

The matrix size we have selected for our problem model ranges from 200 to 9289. This is used to express small, medium and large problem sizes.

Problem Name	Model Numbers	Non zeros Upper	Sparsity %
200	1 - 12	610	1.52%
300	13	1203	1.33%
2352	14-18	16556	0.29%
2352A	19-30	55642	1.00%
3516	31-36	59741	0.48%
3516A	37-47	81390	0.65%
9289	48-50	250689	0.29%

Table 11: Characteristics of the data models used in simulation showing sparsity ratios.

The following figures show some of the problem models used in our research.

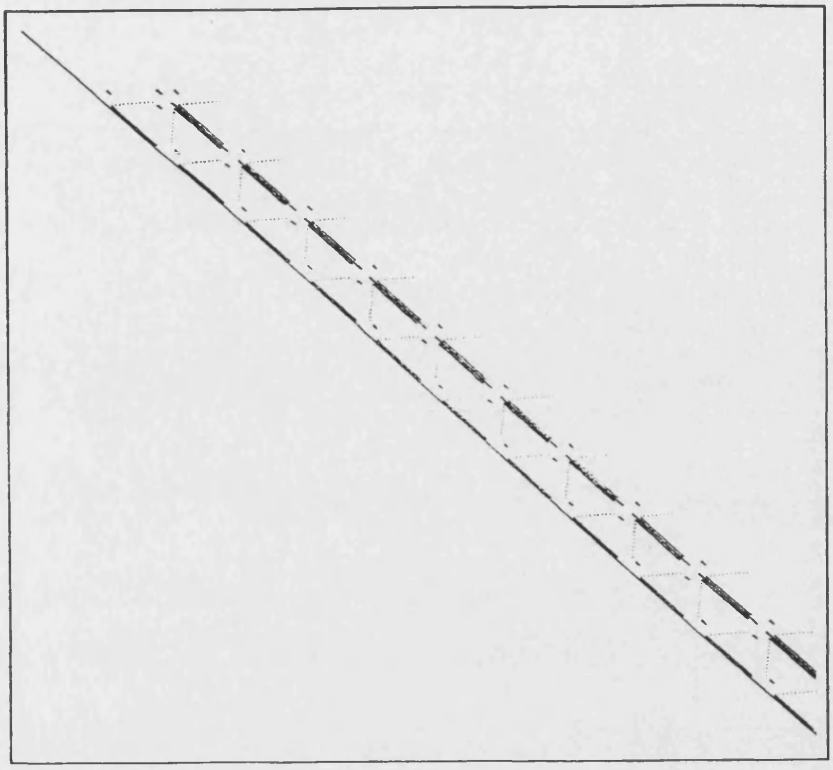


Figure 34: Element distribution in Problem 2352.

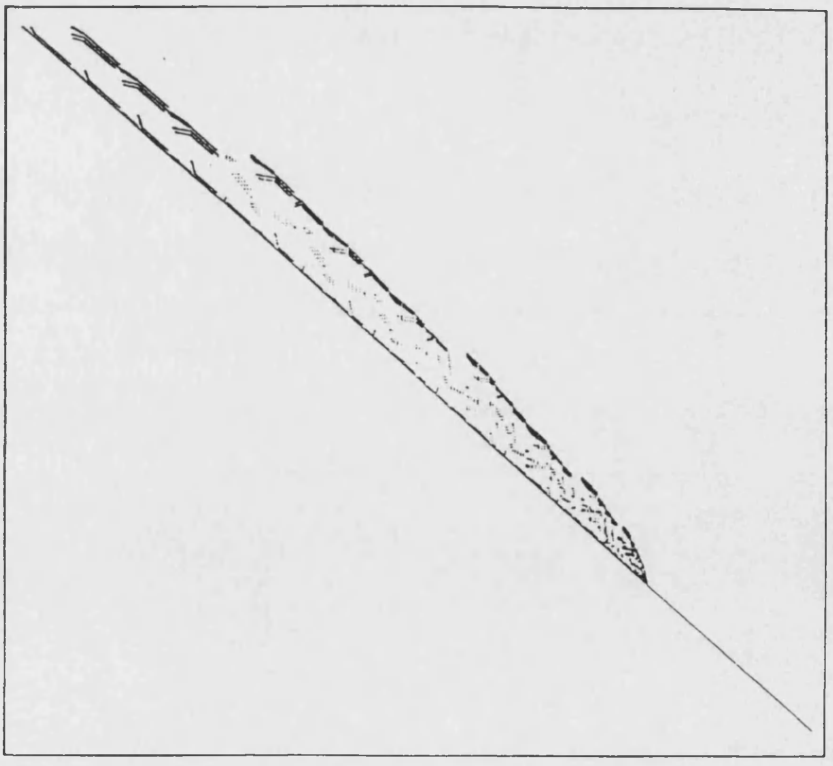


Figure 35: Element distribution in Problem 2352 with renumbering.

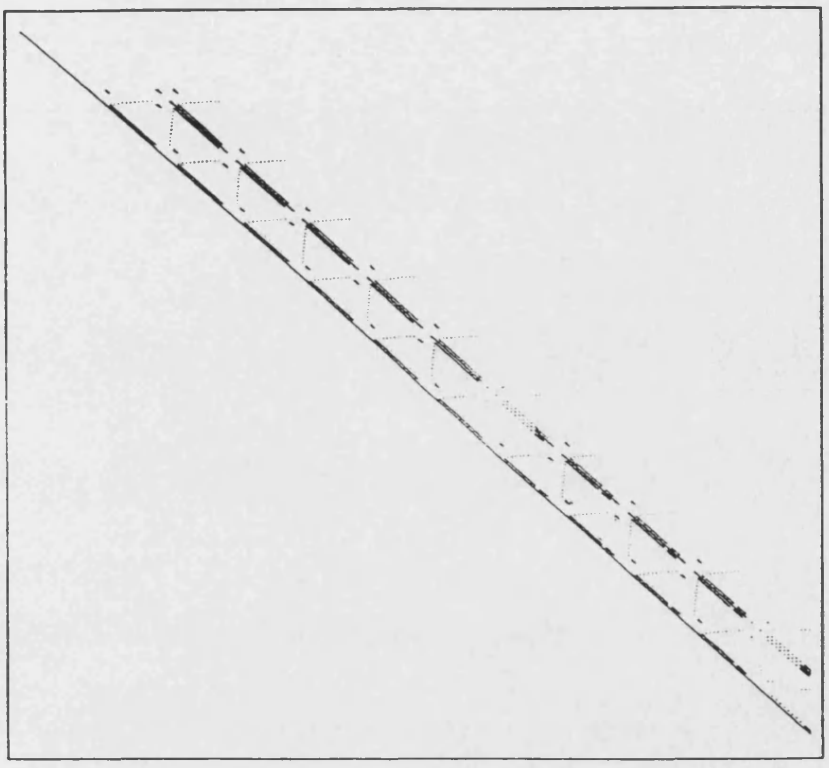


Figure 36: Element distribution in Problem 2352A.

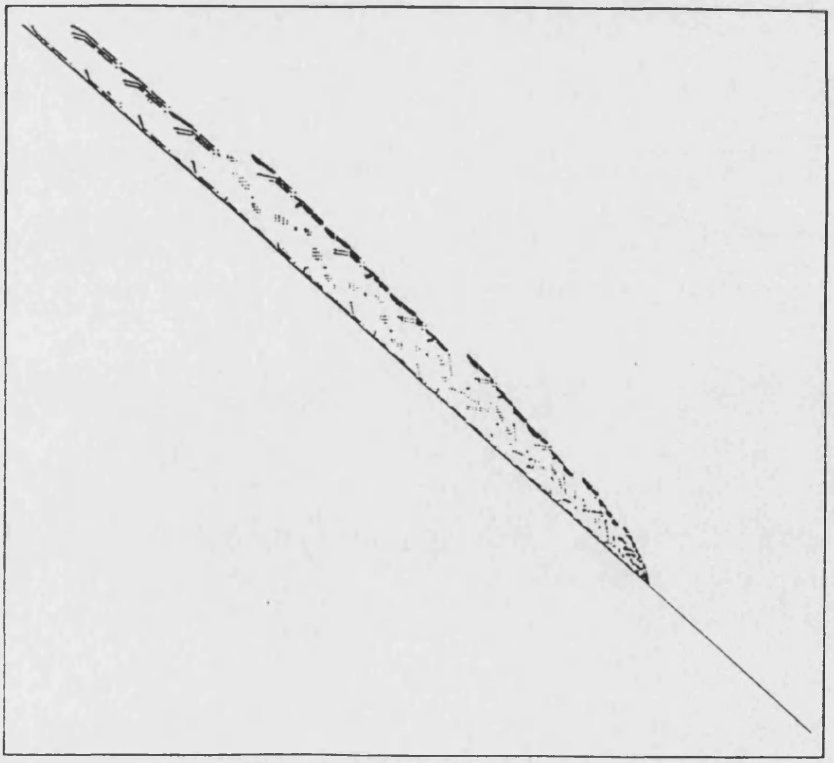


Figure 37: Element distribution in Problem 2352A with with renumbering.

7.3 Backsubstitution Algorithm

The arithmetic operations involved in one iteration of the ICCG algorithm are grouped into two sets. The first set of steps can be easily programmed using vector operations, T_{VEC}. The second set of steps involve the forward- and backsubstitution algorithms, T_{SUB}. Thus the total execution time for a single iteration of the ICCG is $T_{VEC} + T_{SUB}$.

In a parallel implementation of ICCG, T_{VEC} will benefit vector processing capabilities, whereas T_{SUB} will not benefit from such architectures. This is due to the zero element in the matrix used. Thus, T_{SUB} is the potential member for parallel execution improvements.

The time needed to perform a complete matrix preconditioning action which involves a forward- and backsubstitution steps is improved if the scheduling method utilizes the sparsity structure of the matrix used.

To simulate the Backsubstitution algorithm it is necessary to identify and order computation tasks and to schedule processors to perform the tasks (i.e., to partition the problem). There is a tradeoff involved in selecting the level to which computations are decomposed. Defining computation tasks at a low-level (e.g., single arithmetic operation level) exposes most of the problem's potential parallelism; however, it creates a large number of tasks which complicates scheduling and introduces high amount of interprocess communications. Defining tasks at a high level can reduce the parallelism available in the problem; but it results in fewer tasks which simplifies scheduling.

We have identified two types of matrix elements in the backsubstitution algorithm network. The two type of nodes in the network are related to the matrix element position in the matrix to be simulated. Thus for simplicity each

node in the network will have an associated node which performs the same arithmetic operation as on the matrix coefficients.

These two distinct nodes are:

1- Nodes associated with diagonal elements, to be known as diagonal-nodes, and,

2- Nodes associated with row elements not diagonal, to be known as row-nodes.

Diagonal-Nodes:

The number of diagonal nodes in each model is fixed and equals to the matrix dimension. Associated with these nodes is a Divide arithmetic operation. As shown in Figure 37.

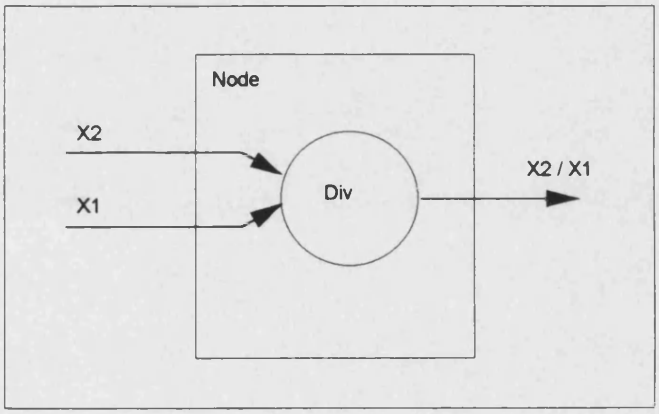


Figure 38: An example of a diagonal element node with its task.

Row-Nodes:

The number of row nodes may differ from one model to another. Their position inside the matrix is identified by the sparsity structure. Associated with this type of nodes is a multiply arithmetic operation.

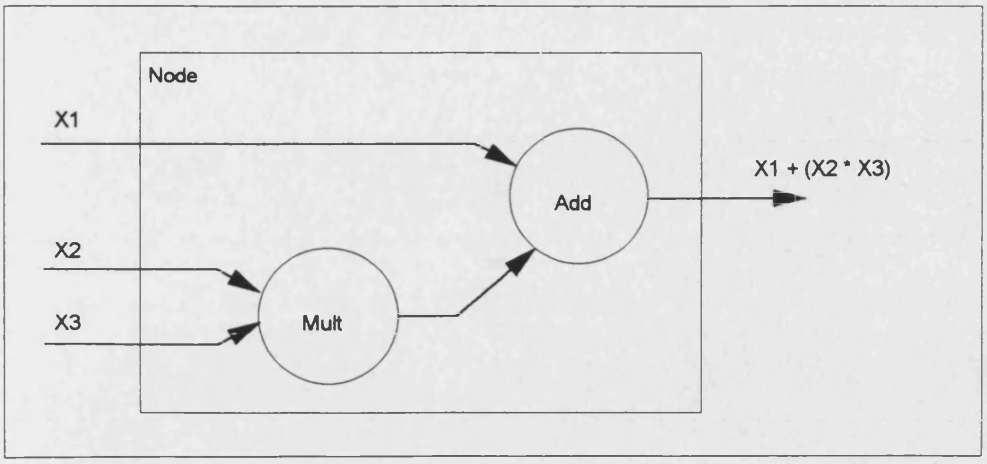


Figure 39: An example of a row element with a Multi-operation task (Add & Mult).

In order to complete the computation of one row, all diagonal elements below this row must be computed first and an extra summation operation of all the row results needs also to be computed. Also one subtract operation is needed to complete the row computations.

7.3.1 Data Dependency in Backsubstitution algorithm

The are several characteristics of the Backsubstitution algorithm which act to restrict the speedup. These include both data dependency among the matrix elements and the communication between the different nodes to transmit intermediate results. Data dependency will restrict the ability to execute the Backsubstitution algorithm nodes in parallel. We have utilized in this research the sparsity structure of the matrix to obtain some degree of parallelisation.

The resultant values of x from a diagonal node has to be transmitted to all the row elements in the same column. These values are needed to complete the computations, but by utilizing the sparsity structure some transmissions could be delayed, thus reducing the communication needs at that particular time.

The cost communication between different nodes to transfer data values will be simulated using a number different values. This would give insight into the effect of interprocessor communication on the performance of the algorithm.

7.3.2 Internal PARASIM presentation of the Network

A parallel program can be regarded as a collection of processes that interact with each other by exchanging messages. For efficient execution of a parallel program it is important that the partitioning (Splitting the program into a number of processes) of a program into processes is done properly. In particular the grain size of processes is important. Processes that perform a lot of computations between successive communications with other processes are said to have a large grain size. In this section we will discuss the different network presentations used. Below are some observations about the network structure:

. The number of arithmetic operations performed in one task. The network may be represented by two or three operations task. Figure 39 shows a task with two arithmetic operations, namely the Add and the Multiply operations. The combination of more than one arithmetic operation into a task definition has improved the simulation response time.

. In the network that represents the backsubstitution algorithm, arithmetic operands of a given *node* are either sourced from the matrix coefficients (constants) or result from another node (variables). The value of constants is known at the start of execution, whereas the variables are computed using nodes.

. During the initial development of the simulation program PARASIM, it was observed that the links representing constants, which sourced from the matrix coefficients were an extra storage and computational burden. It

was decided to remove those communication links from the network structure to improve the simulation response time.

A typical task for non-diagonal element of the matrix is demonstrated in Figure 39. The diagonal elements are associated with a a divide operation to produce the value of the *x*.

The row element interconnection plays an important role in identifying the parallelism available within the network. Figure 40 shows a simple row structure. The row has 5 elements, the elemnt on the left is the diagonal element, other element are numbered form 1 to 4. In this structure the computations starts when the diagonal elements feeding nodes 1 to 4 have produced their *x* result values. The rightmost element will be completed first, then the element on its left and so on.

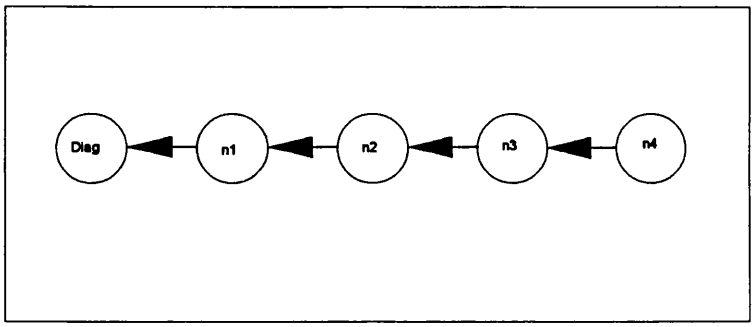


Figure 40: Connections to the diagonal element of each matrix row.

Figure 41 presents a row with 5 elements. The row elements are divided into two groups of 2 nodes each. It is also possible to split the row into a number of groups. In order to complete the computation of the row, an additional Add task is added to sum the results from the groups. Results with row structures have produced a communication bottleneck at the add task. together with the same effect produced by the diagonal node when its data has to feed all elements of that column.

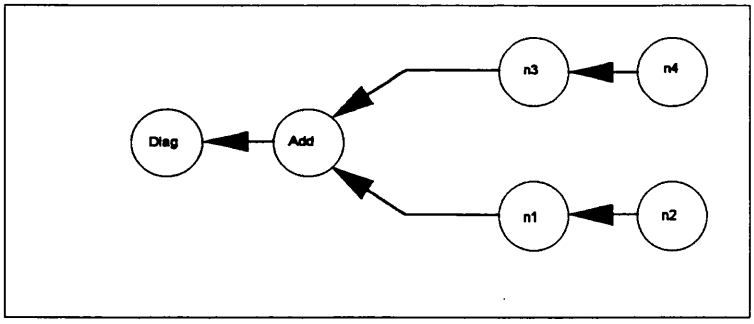


Figure 41: Connecting two matrix elements to a diagonal element by direct connection.

The method we applied to cluster nodes together is depicted in Figure 42. The figure shows a two node and a five node cluster. In this scheme a number of nodes are grouped together and considered as one new node.

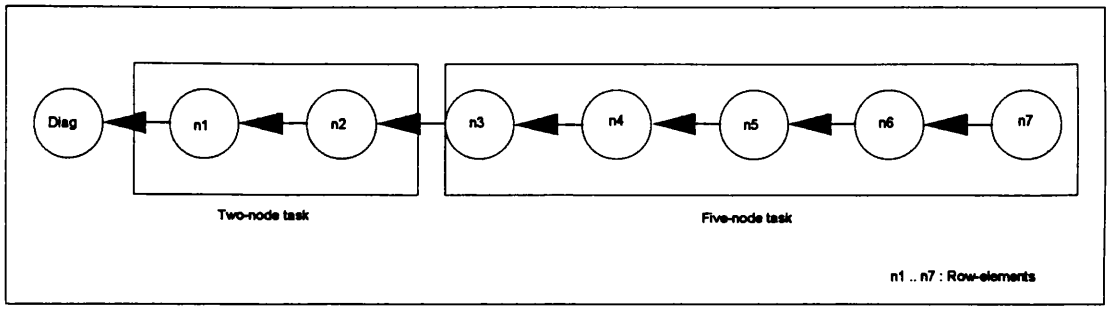


Figure 42: A row with clustered nodes.

7.3.3 Node allocation methods

The simulation program PARASIM is capable of supporting a number of allocation schemes. The allocations schemes are as follows:

1. *Worst-case allocation*: In this allocation scheme all memory accesses are forced to be from remote memory modules. This option would force **PARASIM** to allocate each node to a different processor than that where its input data is coming from. *Worst-case* allocation of nodes is based on the principle of first-ready-first-served. As soon as a processor becomes free it is assigned the next ready node for execution.

2. *Random allocation*: In this case it is assumed that the data is randomly allocated to processors, using the first-ready-first-served method without intervention from the scheduler thereafter.
3. Nodes residing on the critical path are assigned to one processing node namely processor (0). The other non-critical nodes are distributed randomly among the remaining processors.
4. *Best-case allocation* Nodes residing on the critical path are allocated to processor (0), whereas the remaining nodes are allocated considering communication times and processors load balance.

The first 3 schemes proved to produce less speedup results when applied to the same network structures. For the results of this thesis, we have applied only scheme number 4. This is done to study the effect of critical path allocation in the scheduling of nodes.

7.3.4 Time and Granularity of Tasks

In order to investigate the effect of the computational time cost on the execution of a model, we have experimented with different values of time cost for arithmetic operations and communication costs.

A number of task execution times are investigated in the production of simulation results. The instruction and time taken by different tasks describing the network, for example the Add and Multiply times were changed. The time for each arithmetic operation is shown in Table 12. These times are for nodes with single arithmetic operation, four groups of values were used. These values are used as a representation of diffent types of architecture speeds.

Operation	Group 1	Group 2	Group 3	Group 4
ADD	4	3	6	8
SUB	4	3	6	8
MULT	8	4	11	12
DIV	8	4	11	12
Model	3, 19, 20, 37	1, 2	4, 5, 21, 38,	6

Table 12: Single Arithmetic task operation timing and Models used.

Models that use group 1 are 3,19,20,37 and models using group 2 are 1,2 only and models using group 3 are 4, 5, 21, and 38, Group 4 is used only in model 6.

Within each matrix structure we have changed the values of task times using the above mentioned four groups. We have also clustered taks that represent matrix element to increase the granularity of the network, as depicted in Figure 42. The simple row interconnection shown in Figure 40 is also used with single tasks node models. Table 13 shows the number of tasks used in Multi-task nodes together with the model number in which it is used.

Number of Nodes	Time Units	Models
2	20	7, 8, 14, 15, 22, 23, 24, 31, 32, 39, 40 and 48
5	50	9, 10, 14, 16, 25, 26, 33, 34, 41, 42, 43, 44 and 49
10	100	11, 12, 17, 18, 28, 29, 30, 35, 36, 45, 46, 47 and 50

Table 13: Multiple Arithmetic task operation timing.

A network may contain either poor or excessive parallelism. Poor parallelism arises in networks whose nodes are interconnected with other in a manner that forces sequential execution of the network. The other problem is the excess in parallelism, which occurs when the network has a very large number of parallel tasks. In such cases, the computations is said to be *fine-grained.* Fine-grained networks with a large number of interconnections cause degradation in performance due to excessive communication.

7.4 Simulation Results for Uniprocessors

This section discusses the uniprocessor execution results for all the models in our simulation. The execution time on a uniprocessor will be used to find the speedup obtained on the model when it is executed on a multiprocessor architecture. It is used as a reference value. In our calculations of the uniprocessor exection time, we have included only one memory access type. All the operands and constants are assumed to reside in local memory, thus only this memory access was included.

When the network is simulated on a multiprocessor system there are a number of overheads that are encountered, for example, interprocessor communication cost, memory contention, synchronization and scheduling overheads. This section first discusses the different formats and structures of the network used, and then the uniprocessor execution speed of each model.

Each entry of the models contains information on the network structure used to represent the matrix in simulation. **No** (Model Number) is used to reference to each model in the results. The other column information are explain in detail below. Table 14 identifies the label headings for the tables given below. Table 15 to Table 21 give the network structure information.

The **time** column in the following tables is a breakdown of the tasks times used in the network structure, they are separated by "/". The entry for two task models, such as **4/3**, means that the Div and Mult operation takes 4 *ut* to complete, where as the Add and Sub operation takes only 3 *ut* to complete. A three task networkm such as **11/6/20**, the first two numbers will have the same meaning as before and the third means that a multitask of size 20 *ut* is used in the construction of the network.

Label	Description
No	The model number which is unique for each model used in the simulation. It ranges from 1 to 50.
size	The matrix size.
nzero	Non zero elements of the matrix.
Tasks	Number of tasks used to represent the model
time	The time for each task operation in time units. Each task time is separated by a "/".
Nodes	Number of nodes in the model
Links	Number of communication links.
T_{all}	Total execution time on a uniprocessor system in time units.
T_{cpa}	The length of the critical path nodes in time units.

Table 14: Field description for table headings for tables 16 to 22.

The entry **11/6/20/100** will be interpreted as follows: The network has 4 different types of tasks to represent the backsubstitution algorithm. Single tasks node are 11 and 6, where as multi-task node are 20 and 100.

No	size	nzero	Tasks	time	Nodes	Links	Tall	Tcpa
1	200	610	2	4/3	603	1,019	4,690	95
2	200	610	2	4/3	609	1,019	4,690	143
3	200	610	2	8/4	609	1,019	5,100	420
4	200	610	2	11/6	609	1,019	6,730	118
5	200	610	2	11/6	609	1,019	6,730	191
6	200	610	2	12/8	609	1,019	7,540	220
7	200	610	3	11/6/20	449	858	6,247	190
8	200	610	3	11/6/20	451	860	6,253	225
9	200	610	4	11/6/20/50	411	820	6,196	224
10	200	610	4	11/6/20/50	423	832	6,223	237
11	200	610	4	11/6/20/100	451	860	6,253	225
12	200	610	4	11/6/20/100	449	858	6,247	190

Table 15: Network information for models 1 to 12.

No	size	nzero	Tasks	time	Nodes	Links	Tall	Tcpa
13	300	1,203	3	11/6/20	817	1,719	12,681	470

Table 16: Network information for model 13.

17 shows the characteristics of model 13, which is represented by a 3 types of tasks. One of these tasks combine two matrix elements. It has 817 nodes, and requires 1719 communication transfer operations to complete its execution. The model has a uniprocessor execution time of 12681 *ut*. The critical path in the network requires 470 *ut* to complete.

No	size	nzero	Tasks	time	Nodes	Links	Tall
14	2,352	16,556	3	11/6/20	9,807	24,010	180,869
15	2,352	16,556	3	11/6/20	9,852	23,964	180,094
16	2,352	16,556	4	11/6/20/50	6,133	20,336	176,981
14	2,352	16,556	4	11/6/20/50	6,303	20,415	176,443
17	2,352	16,556	4	11/6/20/100	6,223	20,426	177,285
18	2,352	16,556	4	11/6/20/100	6,916	21,028	177,158

Table 17: Network information for models 14 to 18.

No	size	nzero	Tasks	time	Nodes	Links	Tall	Tcpa
19	2,352	22,642	2	8/4	22,641	42,931	214,660	3,412
20	2,352	22,642	2	8/4	22,641	42,931	214,660	18,572
21	2,352	22,642	2	11/6	22,641	42,931	280,234	26,054
22	2,352	22,642	3	11/6/20	13,124	33,413	251,680	27,167
23	2,352	22,642	3	11/6/20	13,231	33,520	252,001	8,207
24	2,352	22,642	3	11/6/20	12,995	33,284	251,293	6,333
25	2,352	22,642	4	11/6/20/50	8,474	28,763	247,030	54,167
26	2,352	22,642	4	11/6/20/50	8,540	28,829	246,913	15,399
27	2,352	22,642	4	11/6/20/50	7,453	27,742	245,488	11,246
28	2,352	22,642	4	11/6/20/100	7,455	27,744	245,753	11,200
29	2,352	22,642	4	11/6/20/100	7,444	27,733	246,000	95,707
30	2,352	22,642	4	11/6/20/100	8,531	28,820	247,301	20,107

Table 18: Network information for models 19 to 30.

No	size	nzero	Tasks	time	Nodes	Links	Tall	Tcpa
31	3,516	59,741	3	11/6/20	32,384	88,608	673,466	12,335
32	3,516	59,741	3	11/6/20	32,247	88,106	669,405	17,109
33	3,516	59,741	4	11/6/20/50	16,941	73,165	657,653	23,307
34	3,516	59,741	4	11/6/20/50	16,881	72,740	653,697	72,740
35	3,516	59,741	4	11/6/20/100	14,412	70,636	655,494	42,513
36	3,516	59,741	4	11/6/20/100	14,959	70,818	652,117	58,663

Table 19: Network information for models 31 to 47.

No	size	nzero	Tasks	time	Nodes	Links	Tall	Tcpa
37	3,516	81,390	2	8/4	81,389	159,263	796,320	21,200
38	3,516	81,390	2	11/6	81,389	159,263	1,306,974	29,751
39	3,516	81,390	3	11/6/20	43,190	121,063	922,374	3,585
40	3,516	81,390	3	11/6/20	43,357	121,230	922,875	18,707
41	3,516	81,390	4	11/6/20/50	21,674	99,547	900,774	63,127
42	3,516	81,390	4	11/6/20/50	21,378	99,251	900,303	7,539
43	3,516	81,390	4	11/6/20/50	21,814	99,687	900,903	35,739
44	3,516	81,390	4	11/6/20/50	16,881	72,740	635,697	32,246
45	3,516	81,390	4	11/6/20/100	16,967	94,840	896,241	113,409
46	3,516	81,390	4	11/6/20/100	17,166	95,039	896,350	20,451
47	3,516	81,390	4	11/6/20/100	17,493	95,366	897,011	63,261

Table 20: Network information for models 37 to 47.

No	size	nzero	Tasks	time	Nodes	Links	Tall	Tcpa
48	9,289	250,689	3	11/6/20	132,142	373,541	2,847,582	37,178
49	9,289	250,689	4	11/6/20/50	64,495	305,894	2,778,816	70,894
50	9,289	250,689	4	11/6/20/100	50,462	291,861	2,765,902	120,342

Table 21: Network information for models 48 to 50.

The serial execution time for each model used in these results are shown in from Figure 43 to Figure 45. The overheads discussed earlier in this section are not included in the computations. This is due to the nature of a single processor execution. These results will form the basis for all the speedup numbers given in this thesis.

An important factor that should be noted when using the uniprocessor execution time, is the fact that all register and memory reference are local to the executing processor, and for the sake of discussion is always assumed to be 1 *ut*.

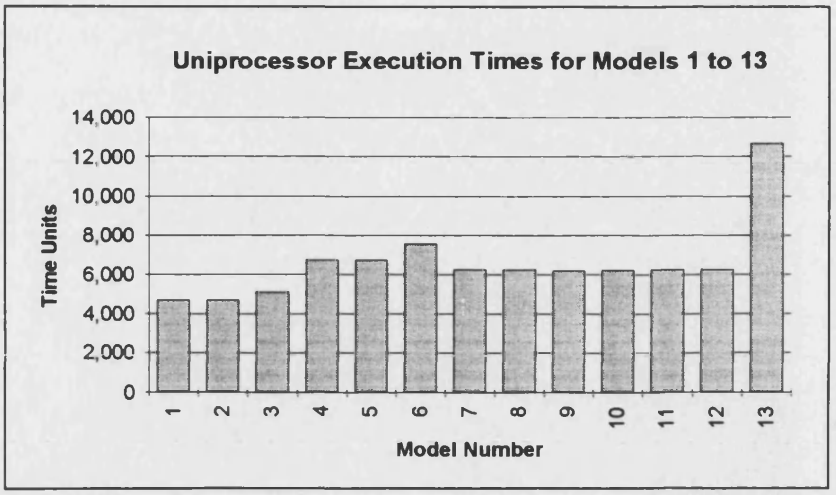


Figure 43: Serial uniprocessor execution time in time units for models 1 to 13.

Figure 43 shows the uniprocessor execution time for models 1 to 13 used in the simulation. Models 1 to 12 represent a matrix of size 200, whereas model 13 represents a matrix of size 300. Model 13 has a large uniprocessor execution time compared to the others because it has 817 nodes to represent it. The number of nodes for matrix 200 range from 449 of model 7 to 609 of model 2.

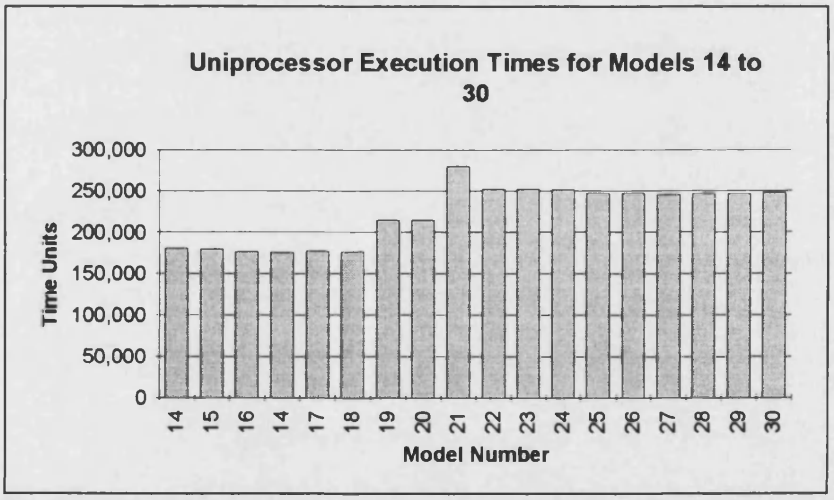


Figure 44: Serial uniprocessor execution time in time units for models 14 to 30.

In Figure 44 models 14 to 18 represent matrix dimension of 2352, but models 19 to 30 represent a matrix with same dimension but it has more elements thus, a dense structure. In this case the time needed to complete its

execution must be more. It is worth noting that model 19 and 20 have less execution time than the other in the same model, this is due to the selected task time to represent the network. We have used a task time of 8 *ut* for Div and Mult, and only 4 *ut* for Add and Sub.

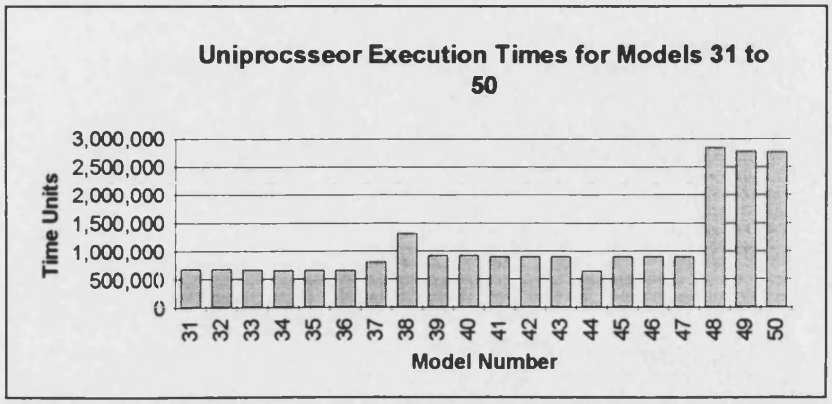


Figure 45: Serial uniprocessor execution time in time units for models 31 to 50.

In Figure 45 model 39 shows an increase in the uniprocessor execution time over the other models (37 to 47). This is due to both the large number of nodes in the network and the tasks times used to compute this time. Models 48 to 50 are of a large matrix problem of size 9289 and number of non zero element 250689.

The following figures from Figure 46 to Figure 48, show the percentage of the critical path time duration compared to the uniprocessor execution time $\frac{T_{CPA}}{T_{ALL}}$ for all the models (1 to 50).

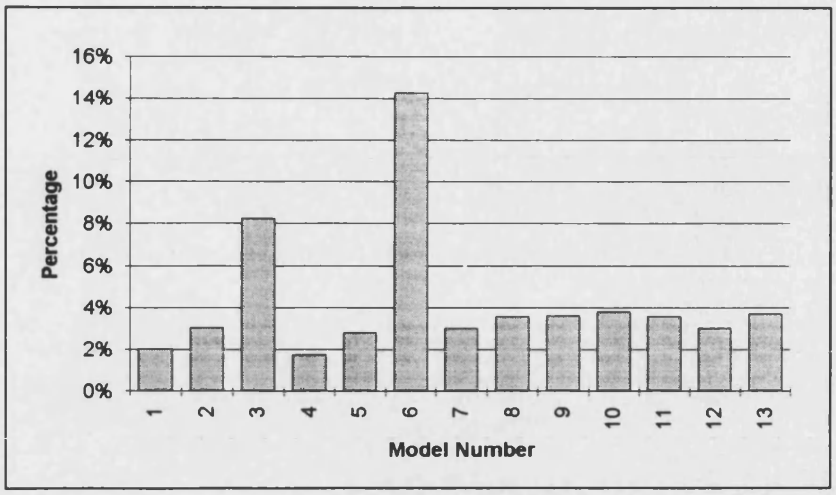


Figure 46: Percentage of critical path time to the uniprocessor execution time for models 1 to 13.

This information is useful in scheduling and balancing of the tasks among the processing nodes. It is necessary to take into account the length of the critical path when the remaining nodes are distributed among the processing nodes.

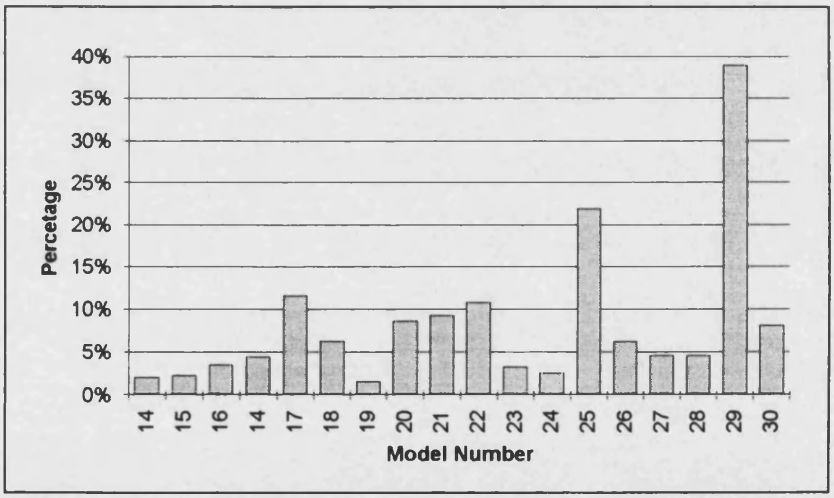


Figure 47: Percentage of critical path time to the uniprocessor execution time for models 14 to 30.

In order to achieve a balanced load among the processing nodes, each processing node will have a comparable size of computations assigned to it. This is computed by deducting the time spent on the crtitcal path from the overall execution time. This is done because the critical path will be executed on only one processing node.

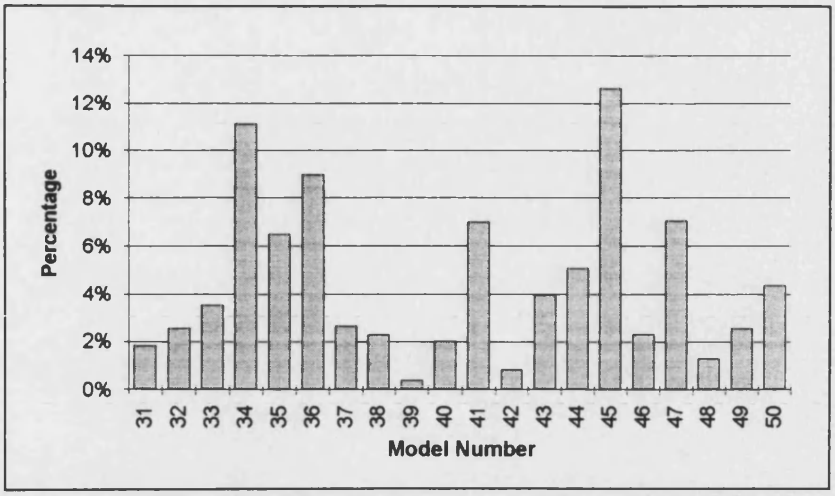


Figure 48: Percentage of critical path time to the uniprocessor execution time for models 31 to 50

7.5 Simulation Results for Multiprocessors

We are going to investigate the effect of the hardware parameters on the speedup results. This section looks only at the different hardware parameters that can be changed by the user to produce a set of simulation results:

1. Architecture of the underlying machine.
2. Interprocessor communication time for the transfer of parameters among different processors.
3. Effect of *broadcast* hardware operation.
4. Effect of Cache architectures.

In order get a quantitative measure for the speedup results, we have computed the average of each speedup result table. The field headings are described in Table 22. The average values are shown in Table 23 for models 1 to 18 and Table 24 for models 19 to 36 and Table 25 for models 37 to 50. All four architectures simulated are included, namely distributed and shared memory and one-to-all broadcast and cache systems respectively.

Field	Meaning
Mod	Model number of the network used.
Arc=1	Distributed memory with complete interconnection.
Arc=2	Shared memory multiprocessor architecture.
Arc=3	Shared memory multiprocessor architecture with one-to-all broadcast mechanism.
Arc=4	Shared memory multiprocessor architecture with cache mechanism.

Table 22: Field headings for average speedup tables.

There are three types of result tables. In each type the number of processing nodes and the interprocessor communication cost varies. Careshould be in comparing values with each other, as they may come from different table formats.

Mod	Arc=1	Arc=2	Arc=3	Arc=4
1	n/a	2.2399	2.2399	3.9688
2	n/a	2.7561	2.7768	3.1332
3	n/a	2.7209	2.8031	2.8922
4	3.9356	3.7369	2.2399	4.1777
5	3.9859	3.5228	3.5703	3.9928
6	7.0555	1.7032	1.7040	n/a
7	3.9522	4.1123	4.1790	5.1695
8	n/a	3.4690	3.5157	3.8891
9	n/a	3.7835	3.8670	4.1581
10	n/a	3.4164	3.4740	4.0224
11	4.2350	3.4690	3.5157	3.8891
12	n/a	3.1003	3.2373	3.9342
13	n/a	2.9993	3.2920	4.0156
14	n/a	2.6860	3.4714	4.0406
15	n/a	2.5192	3.2477	n/a
16	4.3544	2.7755	3.5282	4.1515
17	3.3809	2.6710	3.5011	3.6874
18	4.3165	2.5000	3.4558	3.9818

Table 23: Average speedup results for 4 architectures for models 1 to 18.

Table 23 shows the average speedup results extracted from the results tables of models 1 to 18, showing the average speedup time for each architecture simulated. Similarly, Table 24 shows results of models 19 to 36 and finally Table 25 shows results for models 37 to 50.

Mod	Arc=1	Arc=2	Arc=3	Arc=4
19	n/a	2.1531	2.5081	2.8144
20	n/a	1.9088	2.1385	2.4668
21	3.3030	n/a	n/a	n/a
22	3.3349	2.5168	2.9213	3.5130
23	n/a	2.5973	3.0157	4.0165
24	n/a	2.9278	3.4808	4.8380
25	n/a	2.3312	2.7474	3.2210
26	4.1695	2.4165	3.3828	3.8355
27	4.3379	3.0098	4.6867	5.6112
28	4.3698	2.6970	4.4706	5.1581
29	1.7057	1.9680	2.1425	2.2310
30	3.8527	2.2427	3.0107	3.3697
31	3.8428	2.4936	3.9263	4.2880
32	3.8568	2.4660	4.0049	4.2140
33	4.1994	2.4799	4.5123	4.8277
34	4.2177	2.4394	4.5855	4.8470
35	4.1446	2.3096	4.1357	4.3107
36	3.7347	2.2975	3.8922	4.0735

Table 24: Average speedup results for 4 architectures for models 19 to 36.

Table 24 shows the average speedup results for models 19 to 36. Model 19 to 30 belong to the problem matrix of size 2352 which has 22642 non zero elements. Models 31 to 36 belong to the problem model size 3516 which has 59516 non zero elements.

Mod	Arc=1	Arc=2	Arc=3	Arc=4
37	n/a	1.9407	2.2029	2.2600
38	n/a	2.3865	3.1068	3.1784
39	n/a	3.1259	4.6119	5.0378
40	3.8124	2.4522	4.1027	4.3012
41	3.9302	2.2473	4.5864	4.8108
42	4.1043	2.8925	5.6016	5.9527
43	4.0872	2.5404	4.6492	4.9292
44	4.2177	2.4394	4.5855	6.6332
45	n/a	2.1431	3.9605	4.1282
46	4.2377	2.8324	5.4609	5.8862
47	3.9700	2.3827	4.2068	4.2880
48	4.5406	2.4024	4.0466	4.2418
49	4.0948	2.4379	2.4165	5.1390
50	4.2331	2.2913	4.5170	4.6239

Table 25: Average speedup results for 4 architectures for models 37 to 50.

The above 3 tables give an average of each detailed table presented in Appendix-A. The 4 architectures simulated are shown in separate columns.

Some entries of these tables originated from less entries in the detailed tables.

7.5.1 The effect of the number of Processing Nodes.

The speedup results tables in Appendix-A show the effect of adding Processing Nodes to execute the given model. For each network the processing nodes were varied from 2 to 64 processing nodes allocated to execute the network.

Depending on the architecture used, the increase of processing nodes will improve the speedup results. But this improvement could be eroded by the communication cost and overheads.

7.5.2 The effect of the Interprocessor Communication times

The speedup results tables in Appendix-A show the effect of the interprocessor Communication IPC time. We have varied the value of IPC from 4 to 40 *ut* in distributed memory architectures and from 2 to 32 in shared memory architectures. Increasing the values of IPC always adds more time to complete the execution of the network on the assigned architecture. It is necessary to reduce the time spent in communication to improve the speedup results.

Combined effect of increased PNs and IPC

The number of times needed to synchronize and exchange results increases with the number of nodes (Diagonal or non diagonal elements) being executed. Increasing the number of processing nodes PNs will improve the speedup of the problem, however, as more processing nodes PNs are added, the speedup increases and then begins to actually decrease.

This changeover, from increase to decrease, is a consequence of the fact that the amount of *remote* memory transfers increases (that is memory

residing on remote processing nodes and requires a multiprocessor bus access), thus the effect of bus contention becomes more apparent. The time-shared bus has to serialize the request for data transfers and carry one piece of information every time. More and more time is spent waiting for communication to take place.

7.5.3 Distributed Memory System

The distributed memory architecture used the simulation assumes a fully interconnected model of processing nodes. Refer to section 6.4.1 *Simulated architecture models* for details of the distributed memory architecture.

Both Figure 49 and Figure 50 are speedup results for distributed memory architectures. Model 18 is an example of the medium size matrix and model 50 is the example of a large size matrix. In both figures the increase of the processing nodes will improve the speedup results. It is worth noting that in this architecture there is no effect of bus contention.

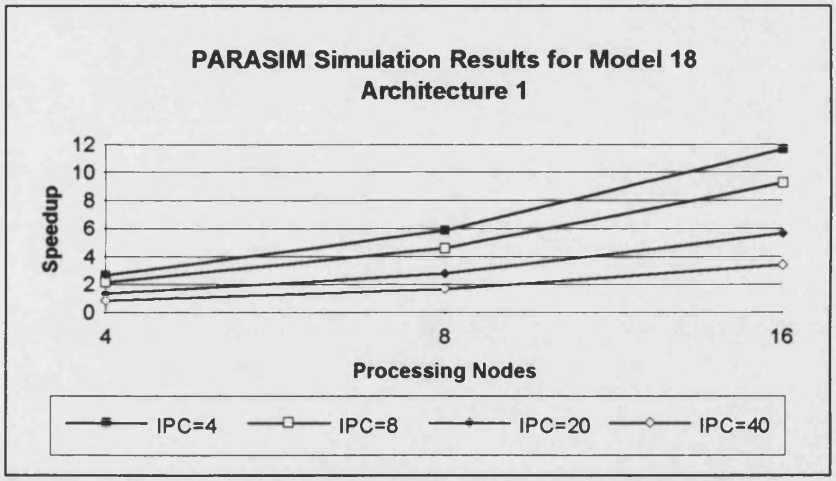


Figure 49: Speedup results for model 18 showing the effect of 4 values of IPC using the distributed memory architecture.

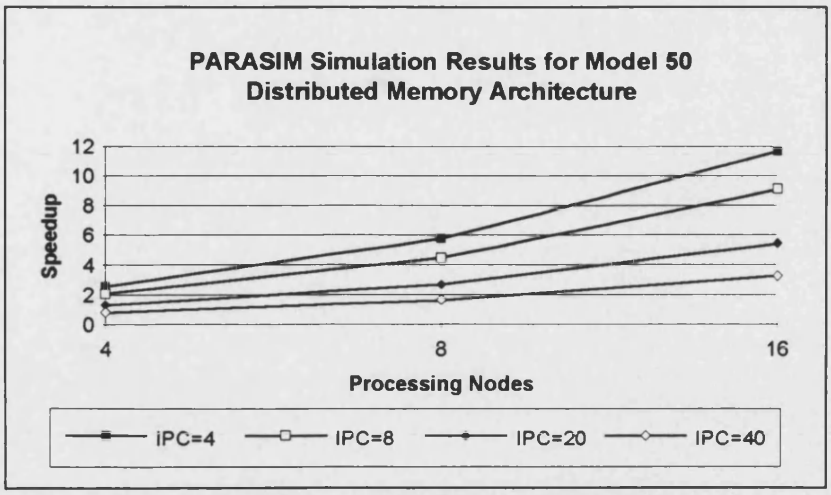


Figure 50: Speedup results for model 50 using Distributed memory architecture.

7.5.4 Shared Memory Systems

The shared memory architecture used in the simulation of the network has the following characteristics:

1- The processing nodes are identical and ranging from 2 to 64 processing nodes.

2- A dedicated single time-shared bus is assumed to connect each processing node to the other in this architecture.

3- As there is no point-to-point communication link between each processing node and the other, the multiprocessor bus is used instead. The bus structure will be activated and it will introduce waiting delays, to allow the serialization of data to the requesting processing nodes.

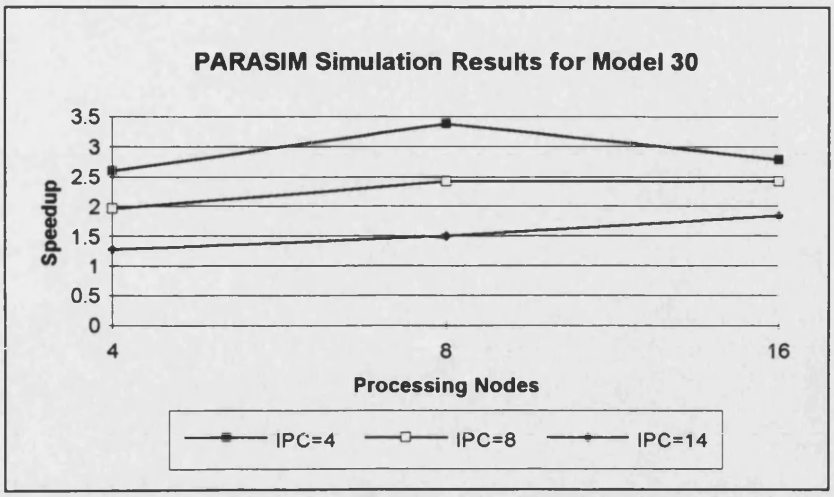


Figure 51: Speedup results for Model 30 showing the effect of 3 values for interprocessor communication using shared memory architecture.

7.5.5 Bus Contention and practical Implementation:

In the implementation of the backsubstitution algorithm on a parallel processor limitations to performance arise in two related areas:

1- Inherent lack of parallelism in some steps of the algorithm; details of this overhead was discussed earlier.

2- Hardware contention problems. These arise due to common utilization of hardware resources, such as the multiprocessor bus structure. In this context the problem is most apparent where processor nodes share common data elements, such as diagonal elements.

7.5.6 One-to-all broadcasting System

One of the communication enhancement features of our simulation program is the one-to-all broadcast mechanism added to the shared memory system architecture. Using this mechanism a result could be sent to all the processing at a reduced cost than each processing node attempting to get hold of the result by itself.

On average the one-to-all broadcast mechanism has improved the speedup results by 20 to 30% over the shared memory architecture.

7.5.7 Shared with Cache mechanism

The cache mechanism feature was added to enhance the performance of the backsubstitution algorithm on multiprocessor systems. In this mechanism the cache logic will look for results that appear on the multiprocessor bus, and makes a copy of these results into the processing node's memory. When the time comes for the processing node to utilize a result from a previous operation, it will find it in the local memory and thus avoiding an expensive multiprocessor bus access. This is done at a cost of a local transfer time.

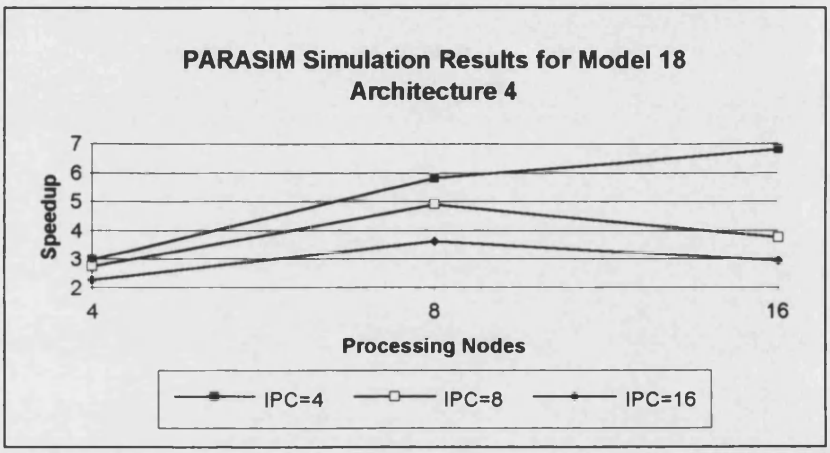


Figure 52: Speedup results for Model 18 showing the effect of 3 values for interprocessor communication using shared memory architecture with cache mechanism.

In Figure 52 the speedup obtained from model 18 is shown. This model has a Multi-task structure of 10 row-elements. When IPC is 4 *ut* the speedup improved, whereas it declined when IPC=8 and IPC=16 and the processing nodes were increased from 8 to 16. The granularity, architecture and the IPC value produced a better speedup results.

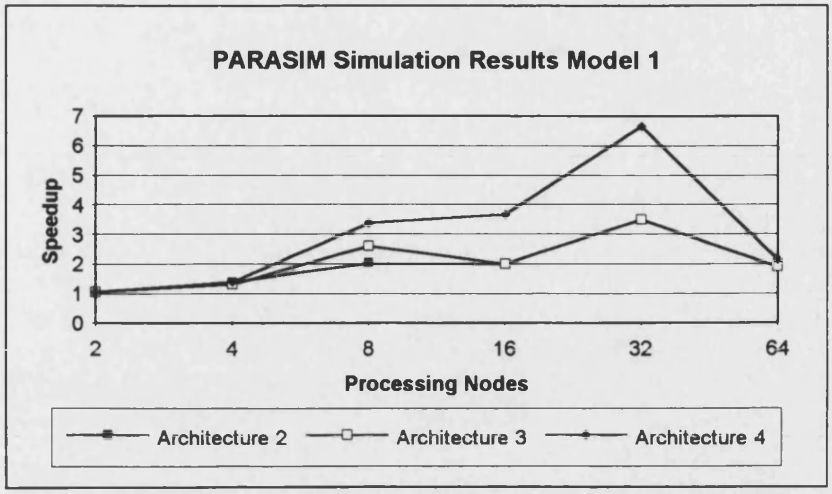


Figure 53: Speedup results for Model 1 showing the effect of 3 architecture types with IPC=8.

In Figure 53 we observe a sudden drop of the speedup when the number of processing nodes is increased from 32 to 64. The increase in number of processing nodes althoughit provides more computing power, but also inroduces delays to the sharing of the bus resources by all processing nodes used.

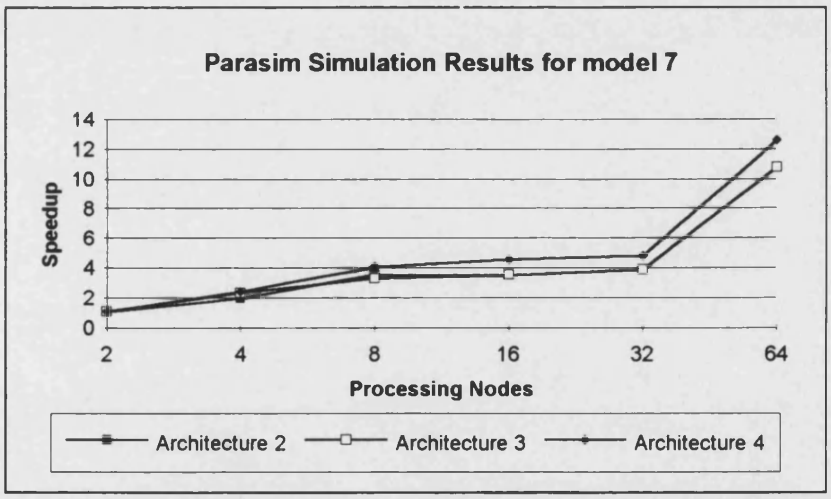


Figure 54: Speedup results for Model 7 showing the effect of 3 architecture types with IPC=8.

In Figure 54 we observe a sudden jump in the speedup results when the number of processing nodes is increased from 32 to 64. This is due to the availability of more processing power. The remote memory transfers which

were channeled through the multiprocessor bus are now local accesses. This will reduce both wait time and makes local access.

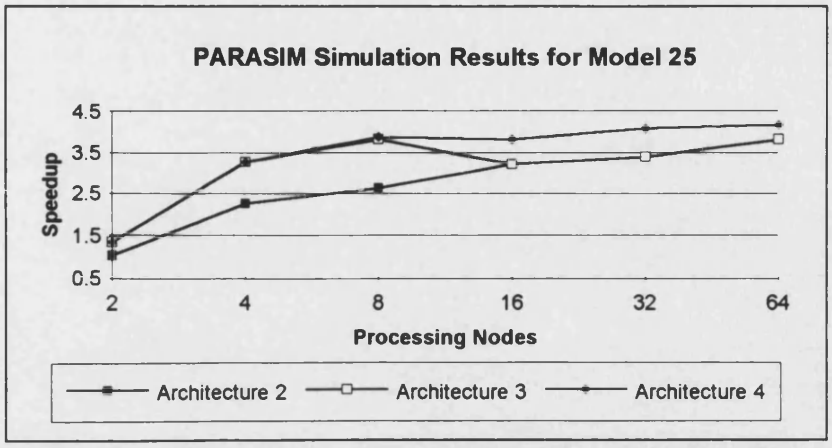


Figure 55: Speedup results for Model 25 showing the effect of 3 architecture types with IPC=8.

In Figure 55, both the shared memory and the broadcast memory lines merge together, when 16, 32 and processing nodes are used. This is due to the crieria of activating the broadcast option. We have used Optimezed broadcast which will allow the node to broadcast its results if it has the feed a number of nodes which is equal to or more than the processing nodes involved in the simulation. Since there are no nodes which will feed more than 15 nodes, then the one-to-all braodcast is inhibited.

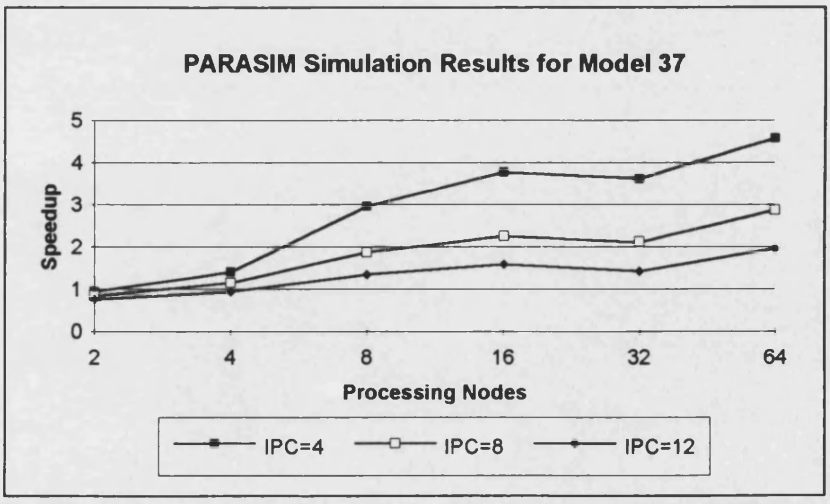


Figure 56: Speedup results for Model 37 showing the effect of 3 values for interprocessor communication using shared memory architecture.

In Figure 56, the speedup increases by introducing more processing power, but reduces slightly with 32 processors. The communication overheads has the major effect in this action.

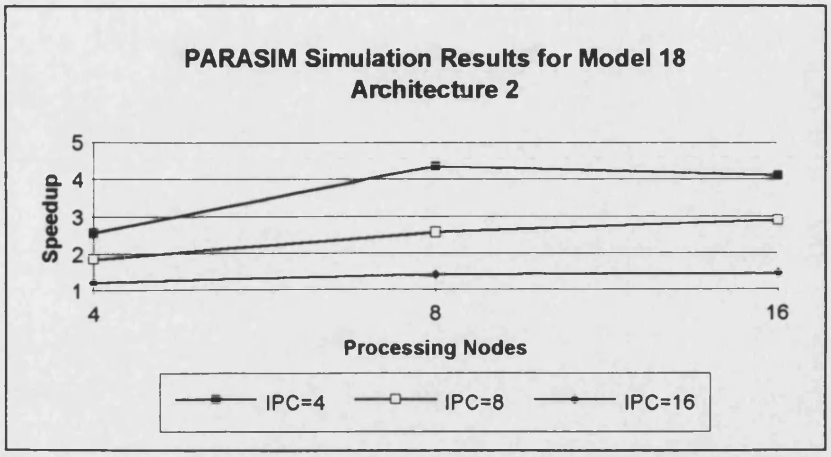


Figure 57: Speedup results for Model 18 showing the effect of 3 values for interprocessor communication using shared memory architecture.

In Figure 57, the speedup results for model 18 are shown for 3 values of interprocessor communication times (namely, 4, 8 and 16). We a smooth increase in speedup with addition of more processing nodes. The interprocessor communication time always reduces the speedup due to the time spent on communication is increased.

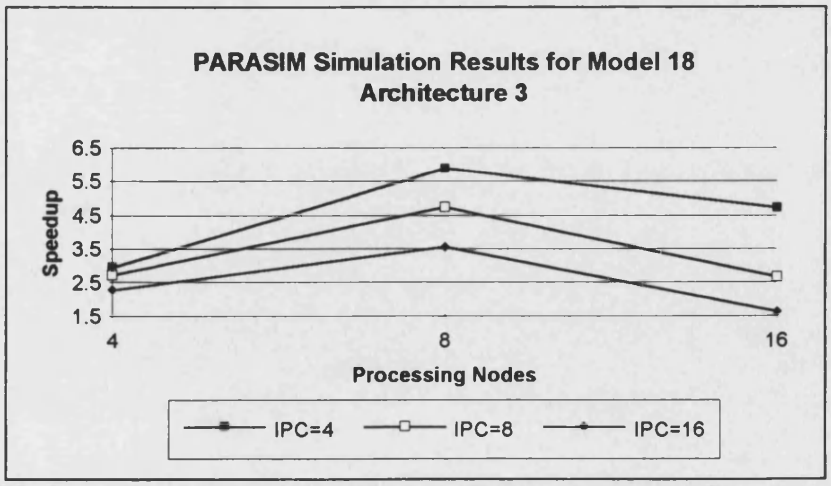


Figure 58: Speedup results for Model 18 showing the effect of 3 values for interprocessor communication using broadcast shared memory architecture.

In Figure 58, results with 8 processing nodes are better than 16 processing, due to the overheads introduced.

7.5.8 Intra-Model parameters

In the previous sections we have investigated parameters that characterized the model, In this section we will broaden our discussion to include the effect of changing the parameters across the models. We will discuss the effect of increasing both the network granularity and the critical path length.

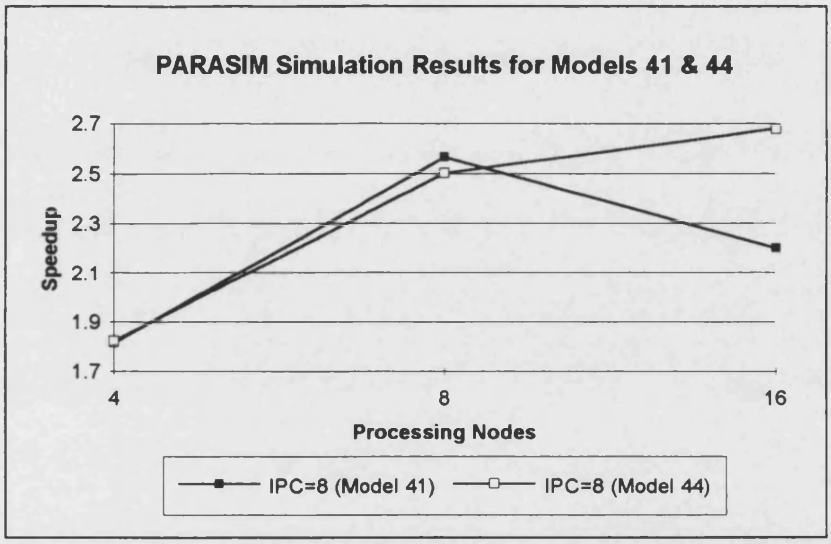


Figure 59: Speedup results for Model 41 & 44 combined showing the effect granularity using architecture type 2.

In Figure 59, two models are represented, namely model 41 and model 44. Both model are for the same matrix, but the network generated has increased the critical path length from 32246 *ut* to 63127 *ut* by reordering the matrix elements. We notice that on 8 processing node systems the model with longer critical path performed better than the model with a short critical path. But this trend has reversed when the number of processing nodes was increased to 16. The speedup of model 41 dropped, whereas the speedup of model 44 increased.

This is attributed to both available processing power and interprocessor communication. Model 41 with a longer critical path utilized the processing power better than model 44 using 8 processing nodes. Model 44 benefits from the spread of computations anong a number of processing nodes.

Definition: Let us define the following function *S* the speedup obtained using a model and an architure.

S(Model number, Architecture, Number of processors, Interprocessor communication)

Where:

Model number: Is the model number of the network used to compute the resultant speedup.

Architecture: The type of architecture used to run the used network, where **1** denoted Distributed memory architecture, and **2** denotes Shared memory architectures, **3** Shared memory with broadcast mechanism and **4** Shared memory with Cache mechanism.

Number of processing Nodes: The number of processing nodes assigned to execute the model.

Interprocessor Communication Time: The cost of communication an opreand to another processing nodes, excluding contention delays.

It is assumed in these results that the local memory reference time is contant in all models and equals 1 *Time Unit.*

To compare the effect of task time between model 1 and model 3, we have found that there is improvement on architectures 2 and 3, and a reduction of 11.0% in the Cache architecture. This is to the communication overhead introduced by Cache.

Parameters	Speedup	Parameters	Speedup	Ratio
S(3,2,8,4)	3.3597	S(1,2,8,4)	3.2323	+3.0%
S(3,3,8,4)	3.5148	S(1,3,8,4)	3.3912	+3.0%
S(3,4,8,4)	3.8375	S(1,4,8,4)	4.2714	-11.0%

Table 26: Comparison of models 1 and 3.

Comparing Models 9 and 11 to study the effect of increasing the granularity of the network. We found that in this small size model no improvement is achieved.

Parameters	Speedup	Parameters	Speedup	Ratio
S(9,2,8,4)	4.9097	S(11,2,8,4)	4.5181	+8.0%
S(9,2,32,4)	7.3478	S(11,2,32,4)	4.1307	+77.0%
S(9,3,8,4)	4.8180	S(11,3,8,4)	4.9745	-4.0%

Table 27: Comparison of models 9 and 11.

Because the completion time for a node with a multi-task structure requires that all the diagonal elements must complete the calculation of all x values involved in the task. Waiting for all diagonal elements to be completed will introduce significant delay to the processing node, and it delays the start of execution of the multi-task node.

We will also compare Models 14 and 15 as an example of large size matrix model to study the effect of granularity. A noticable improvement was found in model 15 which has a higher multi-task nodes.

Parameters	Speedup	Parameters	Speedup	Ratio
S(14,2,8,4)	4.6388	S(15,2,8,4)	4.7665	+2.0%
S(14,2,8,8)	2.7090	S(15,2,8,8)	4.5140	+66.0%

Table 28: Comparison of models 14 and 15.

Another example for the large size matrix model is Model 22 and Model 28. Model 28 has 10 node multi-tasks whereas, model 22 has only 2 node multi-tasks.

Parameters	Speedup	Parameters	Speedup	Ratio
S(22,2,8,4)	4.2755	S(28,2,8,4)	4.9596	+16.0%
S(22,3,8,4)	5.2898	S(28,3,8,4)	6.1264	+15.0%
S(22,4,8,4)	5.2790	S(28,4,8,4)	6.2471	+18.0%

Table 29: Comparison of models 22 and 28.

On large matrix problem models there is an improvement of about +15%, when large multi-task networks are used the represent the same problem.

S(32,2,8,4)	4.5392	S(47,2,8,4)	4.1105	-10.0%
S(32,3,8,4)	5.8222	S(47,3,8,4)	6.1627	+25.0%
S(32,4,8,4)	5.8684	S(47,4,8,4)	6.1293	+4.0%

Table 30: Comparing Model 32 and 47

A reduction in speedup has appeared in model 47 compared to model 32 in shared memory architecture, but an improvement of about 4-5% was obtained using communication enhancement mechanisms.

The effect of increasing critical path length

When the critical path of a network structure is increased by matrix reordering, than a new matrix is formed. It is difficult to compare two different models, because they do not share any characteristic except the original matrix structure.

Let us consider two examples of networks in which the critical path was increased. Model 5 has T_{CPA}= 191 *ut* and model 4 has a T_{CPA}= 118 *ut*. An improvement in performance of +6.0% on shared memory systems and +2.0% on one-to-all broadcast system and +3.0% on Cache systems was achieved.

S(4,2,4,4)	2.3263	S(5,2,4,4)	2.4880	+6.0%
S(4,3,4,4)	2.6005	S(5,3,4,4)	2.6685	+2.0%
S(4,4,4,4)	2.6434	S(5,4,4,4)	2.7402	+3.0%

Table 31: Comparing Models 4 and 5.

Comparing models 39 and 40, which the critical path was increased. Model 39 has T_{CPA}= 3585 *ut* and model 40 has a T_{CPA}= 18707 *ut*. A very small improvement was achieved on communication enhanced archiectures.

S(39,2,8,4)	4.6533	S(40,2,8,4)	4.5178	-3.0%
S(39,3,8,4)	5.7740	S(40,3,8,4)	5.8707	+1.6%
S(39,4,8,4)	5.8190	S(40,4,8,4)	5.8726	+0.9%

Table 32: Comparing Models 4 and 5.

But in general our results show that with the increase of the critical path length, processing power requirements also increase. Models with large

number of elements would benefit from an increase in the critical path, whereas small models tend to decrease in perormance when the critical path is lengthend.

There is delicate balance between the computing architecture, number of processing nodes, and the interprocessing communication delays. It is very difficuly to compare different models together, some models have better performance results with a specific combination of processing ndoes and interprocessor communication. This improved performance is usually due to the match of the problem structure to the architecture used.

7.6 Discussion of results

In summary, the following observations can be made about the simulation results presented in this chapter:

1- It is possible to attain some speedup with the backsubstitution algorithm.

2- Scheduling using the critical path analysis will produce better results.

Three factors limit the actual performance of the parallel system in the simulation the backsubstitution algorithm. First, inherent characteristics of the problem matrix operations and the solution algorithm limit the parallelism that is available in the solution. The critical path is a manifestation of these characteristics; the execution time for a single iteration of the backsubstitution algorithm can't be reduced below the time required to execute the serial chain of operations along its critical path, refer to chapter 2 for the operations of one iteration. Second, the intervals between periods of computation during which processing nodes PNs synchronize their operations, acquire results, and initiate the next iteration cycle represent a system overhead during which there is no useful computation. Third, the time the processing node, allocated to perform critical path computations, spends exchanging variables with others during periods of computation has the effect of increasing the length of the critical path execution time.

Two approaches could be used to reduce the length of the period of data exchange. First, all processing nodes could transfer results concurrently rather than sequentially, with dedicated bus systems (As we have simulated the distributed memory architecture). In addition to allowing the processing nodes to operate in parallel during data exchange, concurrent operation would eliminate overhead associated with exchange of data and use bus bandwidth more efficiently. In the second approach each processing node would transfer its results during the proceeding period of computation when it finishes its computations.

Most processors would perform the transfers during intervals when they would otherwise be idle waiting for the longest-running processor to complete its computations. In general data would be transferred to a buffer in each processing node to prevent overwriting old values that are still in use. Each processing node would move its buffer contents to working storage at the start of the next period of computation. It is not possible to completely eliminate the period of data exchange because all processors need to synchronize their operations before the next iteration is started, to ensure that all variables are current.

Comparison of Results

We have attempted to compare the results of our simulation with results by other authors; this has not been an easy task. The main problem is that the simulations of this type are necessarily complex and published accounts do not give sufficient information to determine how a specific model operates.

7.7 Summary and Conclusions

In summary, the following observations can be made about the simulation results presented in the chapter:

From our results it is clear that an increase in the number of processing node does not automatically lead to a linear speedup. This is mostly due to the effects overheads and interprocessor communications.

The speedup obtained by parallel algorithm is usually dependent upon characteristics of the hardware architecture (such as the interconnection network, the CPU speed and throughput, and the speed of the communication channel) as well as on certain characteristics of the parallel algorithm (such as the degree of concurrency, data interconnections and overheads due to communication, synchronization and redundant work).

Given a parallel architecture and a problem of fixed size, the speedup of the parallel backsubstitution algorithm does not continue to increase with increasing number of processing nodes but tends to saturate or peak at a certain limit. This happens either because the number of processors exceeds the degree of concurrency inherent in the algorithm or because the overheads grow with increasing number of processing nodes.

Chapter 8

Conclusion and Future Work

8.0 Conclusion and Future Work

In this thesis we have explored the use of parallelism to speedup the solution of large systems of linear equations. We have discussed the parallel features of the Incomplete Choleski Conjugate Gradient ICCG algorithm, parallel computing architectures, previous research in the simulation of such architectures, scheduling parallel tasks using critical path analysis, and presented both our simulation program and results. This chapter iterates the main results of the thesis and discusses directions for future research.

8.1 Steps of the investigations

Many **CAD** application programs exist that involve the solution of large systems of linear equations. The solution of these equations is in many cases the most computationally extensive stage of finding a solution. The algorithms for the solution of the equations have in the past been designed to run on sequential machines. In order to benefit from the availability of parallel computing architecture systems, this research aims to identify the parallelism which exists in the solution of large systems of matrices.

The motivation for this research was to investigate the execution of Incomplete Choleski Conjugate Gradient ICCG algorithm on parallel computing systems by finding the computationally intensive parts, and adapting them for parallel execution. We have found that the backsubstitution segment of the ICCG algorithm is that part of the algorithm which requires further investigations.

We have concentrated our research on identifying and investigating possible improvements in the parallel execution of the backsubstitution algorithm. Utilizing the sparsity structure of the matrices used to gain

speedups. However, all the results of this research indicate that qualitatively, some improvement could be attained with parallel execution of the algorithm.

In order to investigate the performance of the backsubstitution algorithm, we had to model two things. The first is the algorithm itself and the second is the underlying architectures to be tested on. Performance results will depend on the scheduling technique used to assign tasks to processors. Let us first discuss how the algorithm was modeled.

Algorithm Model

To simulate the backsubstitution on a multiprocessor architecture it is necessary to identify and order computation tasks and to schedule the processing nodes to perform the allocated tasks. The is a tradeoff involved on selecting the level to which computations are decomposed. This is known as the granularity of the model.

The backsubstitution algorithm steps were represented, in our case, by a data dependency graph. This graph when implemented in the program is known as the *network*. A node in the network may represent one or more arithmetic operations depending on the granularity of the model, and links between nodes represent the data transfer requests. The links map the communication requirements of the network to complete the steps of the algorithm. The structure of the network is discussed in detail in chapter 6 of this thesis.

It is possible to identify the parallelism available within the network, and find the sequence of nodes that take the longest time to execute. This time, known as the critical path time, gives an indication of the parallel structure inside the network.

Once the algorithm was modeled, and it was possible to change a number of parameters which could affect its parallel execution, then the underlying architecture was simulated.

Parallel Architectures

To investigate issues of multiprocessor architectures, a simulation program was designed and implemented in C language. The program PARASIM, based on discrete event simulations, provided the platform to conduct our investigations. Each component of the parallel computer architecture was modeled, such as: the processing node, memory, bus structures and communication capabilities. By varying different hardware features of the system, we were capable of investigating many hardware effects, such as, the speed of executing arithmetic functions, memory access speeds and effect of bus contention. A number of existing and future parallel computers could be mimicked by the simulation program.

The program had an extra capability to schedule the network nodes. This scheduling capability with simulating the underlying architecture will give an insight into the behavior of the algorithm during execution. The scheduling method we have adopted in PARASIM uses the critical path method. This method ensures the least execution time possible for the given network / architecture combination is obtained.

8.2 Conclusions

The simulation program with its scheduling capability allows a number issues to be investigated. The issues we have investigated are related to (i) The model sizes; (ii) The Network; (iii) The architecture simulated.

The different factors we have investigated are:

() Identification of the critical path.

() Interconnections of data inside the network structure;

() The problem of the network granularity;

() The number of processor nodes allocated to solve the problem,

() The effect of increased interprocessor communication values.

() Distributed memory architecture.

() Shared memory using multiprocessor bus structures.

() Effect of one-to-all broadcast mechanism.

() Effect of Cache mechanisms.

Conclusions derived from the above issues are expanded in the following subsections.

- We have used a number of real matrices generated from the CAD program in our investigations that vary in both size and sparsity structures.

- The amount of time spent in the identification of the critical path is comparable to 1 or 2 iterations of the matrix solution, It does two sweeps over the network to identify the critical path.

- In spite of the dependent steps in the execution of the backsubstitution algorithm, we found that by identifying tasks that could be executed in parallel, we could obtain some parallelism during its execution. The amount of speedup available depends on the sparsity structure of the matrix used, i.e., the position and distribution of matrix elements.

- Another important factor which affected the performance of the network, is the data interconnection of the network. Since all elements in the row contribute to the computation of the result associated with that row, a number of communications equal to the number of elements in the row are needed to find the result. This is coupled with the need to feed the result to all elements in the same column, leads to a large amount of interprocessor communications. We have found that by clustering node elements together performance improved in large problem

models. Increasing the granularity of the network provides better result than fine grained models.

- The number of allocated processing nodes to perform the algorithm, will be beneficial upto a certain saturation limit. This limit is governed by the cost of communicating results and exchanging operands.

- The increase of interprocessor communication costs has always led to a decrease in the performance results. The time spent in communication must be reduced as much as possible to improve results. Inherent characteristics of the problem equations account for the loss of nearly 20-30% of the available processing power.

- We have simulated a distributed memory architecture with fully interconnected structure. This structure will allow communication between any two processing nodes. Although this produced good results, the architecture is impractical and not cost effective.

- The shared memory architecture we have simulated, assumes that all processing nodes are connected using a single multiprocessor time-shared bus. Using this bus structure, requests from processing nodes of operands has to be serialized and the requesting processing node will have to wait until the bus is free. This introduces time delay overheads to the system. These overheads affect the performance of the system.

  As explained earlier, the algorithm has a large number of data transfers which would introduce communication overheads. The results obtained from shared memory systems are less than those of distributed memory.

- Since data transfers consume a lot computing time, any method which reduces the time spent in acquiring the operands will be reflected as improvement in speedup. We have experimented with two forms of communication enhancements. The first being the one-to-all broadcast and the second is cache mechanism.

- One-to-all broadcasting is a feature which could be implemented in shared memory architectures. In this mechanism data could be sent from one processing node to all the processing nodes on the same bus using a single transfer operation, saving each processing node time to acquire its own copy of the operand.

  Our simulation results show that a considerable gain will be obtained by broadcasting the results of the diagonal elements.

- Cache mechanism to enhance communication of operands will offer a good hardware structure for improvrd system performance.

In this thesis we have explored the use of parallelism to speedup the execution of the Incomplete Choleski Conjugate Gradient ICCG algorithm. We have discussed the sources of parallelism available within the algorithm and applied an intelligent scheduling method to attain the best speedup possible.

8.2 Directions for Future research

Although many issues regarding the use of parallelism in ICCG have been addressed in this thesis, many more remain to be addressed. This section discusses some possible directions in future work.

- To incorporate the scheduling and critical path identification steps into a code generation program which will generate the necessary code for

each processing node to solve the matrix structure. Only operands will be sent to their assigned processing nodes for execution.

- In the area of **teaching** parallel architectures and algorithms it is extremely useful to a general simulation program with scheduling capabilities to assist students in understanding the concepts. Evaluating different multiprocessor architectures is one application for the simulation program. Update to CAD package and improved user interface, to be used as a teaching tool.

- An obvious direction for further work is **to implement** the scheduling mechanism proposed in this thesis on an actual multiprocessor system. Such an implementation will bring many interesting issues to light, and a running parallel implementation of will certainly encourage further investigations. Test other types of algorithms and obtain performance results.

- The execution time, and hence the response time of the simulation program, can still be **improved**. Improvement of the program's response time and further investigations into new and faster programming techniques are needed. Also better interaction with the user would facilitate dealing with the different program needs.

- Scheduling parallel tasks will benefit from utilizing sub-critical paths in the network. Adding this feature and investigating the speedup results will provide better insights into the network.

Part FIVE

Related information and Appendix

This part contain the detailed speedup tables of the simulation results using PARASIM and a list of its important procedures.

Appendix A: Speedup tables from

PARASIM

Note:

C\P: indicates that the columns are the number of processing nodes, and rows repesent the interprocessor communication time.

Architecture type 1:
Distributed memory architecture (Fully interconnected)

Architecture type 2:
Shared memory architecture

Architecture type 3:
Shared memory architecture with one-to-all braodcast

Architecture type 4:
Shared memory architecture with Cache mechanism

Field	Description
No	Model number
Size	Sparse matrix dimension
N_{ZERO}	Number of non-zero elements in the sparse matrix
Time	Task times
Tasks	Number of tasks
Nodes	Number of *nodes* in the network
Links	Number of *Links* in the network
T_{ALL}	Uniprocessor execution time
T_{CPA}	Critical path length

Model 200
[Table 1]
[New Run] Matrix: size=200 nzero=610 Density=0.025500
[Generate] non-Diagonal to Diagonal
[Generate] Time: sub=3, div=4, mul=4

PARASIM Simulation results
Matrix size=200, Non-zero=610, Density=0.025500
Network: Nodes=609 Links=1019 Tasks=10
Timing: T_{all}=4690 T_{cpa}=95 T_{local}=1
Best Speedup=49.37, CPA On

Architecture type: 2.

C\P	2	4	8	16	32	64
2	1.0686	1.6846	3.9983	5.8698	9.8117	7.2713
4	1.0623	1.5738	3.2323	4.0085	6.2450	3.9445
8	1.0499	1.3962	2.0374	1.9983	3.5105	1.9081
12	1.0378	1.3282	1.5317	1.2427	2.5027	1.1274
20	1.0145	0.9698	1.0438	0.7632	1.5540	0.5764
32	0.9814	0.7822	0.7158	0.4811	0.9907	0.3202

Architecture type: 3.

C\P	2	4	8	16	32	64
2	1.0725	1.7435	4.2290	5.8698	9.8117	7.2713
4	1.0700	1.5802	3.3912	4.0085	6.2450	3.9445
8	1.0652	1.3123	2.6041	1.9983	3.5105	1.9081
12	1.0604	1.2109	1.9420	1.2427	2.5027	1.1274
20	1.0509	1.0471	1.3693	0.7632	1.5540	0.5764
32	1.0369	0.5802	0.9595	0.4811	0.9907	0.3202

Architecture type: 4.

C\P	2	4	8	16	32	64
2	1.0725	1.8320	4.9525	8.4657	14.4753	21.0314
4	1.0700	1.6514	4.2714	7.5890	11.3012	12.4403
8	1.0652	1.3996	3.3838	3.6755	6.6619	2.1613
12	1.0604	1.0299	2.4516	3.1455	5.7265	4.0889
20	1.0509	0.9139	1.8327	1.8171	3.1561	1.2970
32	1.0369	0.5667	1.3174	1.1879	1.9019	0.7963

[Table 2]
[New Run] Matrix: size=200 nzero=610 Density=0.025500
[Generate] Diagonal to all elements in the column
[Generate] Time: sub=3, div=4, mul=4

PARASIM Simulation results
Matrix size=200, Non-zero=610, Density=0.025500
Network: Nodes=609 Links=1019 Tasks=11
Timing: T_{all}=4690 T_{cpa}=143 T_{local}=1
Best Speedup=32.80, CPA On

Architecture type: 2.

C\P	2	4	8	16	32	64
2	1.0844	1.6864	3.9018	6.0988	10.6591	10.0860
4	1.0799	1.5744	2.9627	4.3547	6.4869	7.5281
8	1.0710	1.4435	2.0771	2.7220	3.5263	4.9947
12	1.0623	1.2811	1.6526	2.0051	2.5080	3.6842
20	1.0452	0.9845	1.1111	0.9970	1.5090	2.3941
32	1.0207	0.7277	0.7336	0.6408	0.9603	1.5659

Architecture type: 3.

C\P	2	4	8	16	32	64
2	1.0859	1.8508	3.8223	6.0988	10.6591	10.0860
4	1.0829	1.6925	3.0594	4.3547	6.4869	7.5281
8	1.0769	1.4592	2.3497	2.7220	3.5263	4.9947
12	1.0710	1.1910	1.7976	2.0051	2.5080	3.6842
20	1.0594	0.8826	1.3305	0.9970	1.5090	2.3941
32	1.0425	0.6565	0.7344	0.6408	0.9603	1.5659

Architecture type: 4.

C\P	2	4	8	16	32	64
2	1.0859	1.9088	4.3791	7.6260	8.5584	12.0256
4	1.0829	1.7772	3.9018	6.7288	5.3784	7.2601
8	1.0769	1.3369	3.0219	4.3146	5.4282	4.2559
12	1.0710	1.1609	2.1279	3.1625	3.7610	2.9627
20	1.0594	0.9103	1.5686	2.1203	2.5798	1.9151
32	1.0425	0.7280	1.1204	1.5272	1.6178	1.2128

[Table 3]
[New Run] Matrix: size=200 nzero=610 Density=0.0255%
[Generate] Diagonal to all elements in the column
[Generate] Time: mul_sub=8, div=4

PARASIM Simulation results
Matrix size=200, Non-zero=610, Density=0.025500
Network: Nodes=609 Links=1019 Tasks=2
Timing: T_{all}=5100 T_{cpa}=420 T_{local}=1
Best Speedup=12.14, CPA On

Architecture type: 2.

C\P	2	4	8	16	32	64
2	1.1505	1.8021	4.3257	6.0932	10.1190	7.9937
4	1.1336	1.6580	3.3597	4.7398	8.8388	4.7932
8	1.1015	1.4242	2.2860	2.94.80	6.0000	2.8271
12	1.0710	1.2358	1.6372	2.1118	3.7010	1.9984
20	1.0147	0.9586	1.1135	1.3439	2.3416	1.2599
32	0.9408	0.6824	0.7270	0.9028	1.5098	0.8082

Architecture type: 3.

C\P	2	4	8	16	32	64
2	1.1836	1.8195	4.2535	6.0932	10.1190	7.9937
4	1.1781	1.7412	3.5148	4.7398	8.8388	4.7932
8	1.1676	1.6028	2.6494	2.9480	6.0000	2.8271
12	1.1570	1.5013	2.1048	2.1118	3.7010	1.9984
20	1.1364	1.1328	1.4925	1.7826	1.3829	1.8169
32	1.1073	0.8684	0.9334	0.9028	1.5098	0.8082

Architecture type: 4.

C\P	2	4	8	16	32	64
2	1.1836	1.8339	4.3664	7.3699	8.4718	10.8974
4	1.1781	1.7592	3.8375	5.5195	5.4140	8.0315
8	1.1676	1.6404	2.8508	4.0702	3.2797	4.3111
12	1.1570	1.4555	2.1665	2.2126	1.8722	2.9877
20	1.1364	1.1328	1.4925	1.7826	1.3829	1.8169
32	1.1073	0.9681	1.1092	1.2263	0.9148	1.0159

[Table 4]
[New Run] Matrix: size=200 nzero=610 Density=0.025500%
[Generate] Diagonal to all elements in the column
[Generate] Time: mul_sub=11, div=6

PARASIM Simulation results
Matrix size=200, Non-zero=610, Density=0.025500
Network: Nodes=609 Links=1019 Tasks=2
Timing: T_{all}=6730 T_{cpa}=118 T_{local}=1
Best Speedup=57.03, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.4871	5.4406	11.5240
8	1.9507	4.1012	8.6061
20	1.1844	2.3490	4.8698
40	0.7158	1.3721	2.8254

Architecture type: 2.

C\P	2	4	8	16	32	64
2	1.0491	2.7605	5.5851	8.5515	13.4600	15.7611
4	1.0442	2.3263	4.4393	6.0089	8.7516	9.9852
8	1.0346	1.7372	3.1671	4.1162	5.2414	5.8932
12	1.0251	1.3455	2.4944	3.0234	3.6339	4.0887
20	1.0067	0.9480	1.6391	1.9829	2.4917	2.5825
32	0.9803	0.7053	1.0706	1.2681	1.5925	1.7363

Architecture type: 3.

C\P	2	4	8	16	32	64
2	1.0545	2.8971	5.3244	8.5515	13.4600	15.7611
4	1.0535	2.6005	4.5350	6.0089	8.7516	9.9852
8	1.0516	2.0493	2.9009	4.1162	5.2414	5.8932
12	1.0496	1.6630	2.3899	3.0234	3.6339	4.0887
20	1.0457	1.4195	1.2975	1.9829	2.4917	2.5825
32	1.0399	1.0139	0.8868	1.2681	1.5925	1.7363

Architecture type: 4.

C\P	2	4	8	16	32	64
2	1.0545	2.8835	5.7277	9.4522	14.0501	17.9467
4	1.0535	2.6434	4.8874	6.4774	11.4651	12.6266
8	1.0516	2.1766	3.8902	4.4481	6.3252	6.7435
12	1.0496	1.6838	2.6299	3.1850	4.5473	3.7894
20	1.0457	1.3000	1.7720	1.9384	2.7741	2.0663
32	1.0399	0.9132	1.4433	1.2945	2.0525	0.9692

[Table 5]
[New Run] Matrix: size=200 nzero=610 Density=0.025500%
[Generate] New method, Time: mul_sub=11, div=6

PARASIM Simulation results
Matrix size=200, Non-zero=610, Density=0.025500
Network: Nodes=609 Links=1019 Tasks=2
Timing: T_{all}=6730 T_{cpa}=191 T_{local}=1
Best Speedup=35.24, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.5329	5.4671	11.4068
8	1.9673	4.1466	8.6504
20	1.1780	2.4044	5.0149
40	0.7060	1.4068	2.9492

Architecture type: 2.

C\P	2	4	8	16	32	64
2	1.0635	2.8541	5.1809	9.1940	11.4651	14.4421
4	1.0592	2.4880	4.6318	7.1596	8.0119	9.5326
8	1.0506	1.8126	2.8602	4.5596	4.7697	5.5620
12	1.0421	1.3732	2.1787	3.2002	2.4607	3.7682
20	1.0256	0.9861	1.4730	1.6047	1.5739	2.3319
32	1.0018	0.6708	0.9266	1.0163	0.9997	1.4909

Architecture type: 3.

C\P	2	4	8	16	32	64
2	1.0703	2.9235	5.5074	9.1940	11.4651	14.4421
4	1.0689	2.6685	4.2247	7.1596	8.0119	9.5326
8	1.0662	2.1304	3.0675	4.5596	4.7697	5.5620
12	1.0635	1.7678	2.1900	3.2002	2.4607	3.7682
20	1.0582	1.2083	1.4650	1.6047	1.5739	2.3319
32	1.0502	0.8787	0.9791	1.0163	0.9997	1.4909

Architecture type: 4.

C\P	2	4	8	16	32	64
2	1.0703	2.9274	5.5482	9.5869	11.7452	16.6998
4	1.0689	2.7402	3.9495	8.7974	6.6110	12.5560
8	1.0662	2.1710	3.1987	4.5047	4.2703	7.2288
12	1.0635	1.9229	2.4981	3.2670	3.6241	5.7669
20	1.0582	1.2526	1.9240	2.1929	2.4322	3.4637
32	1.0502	0.9002	1.0171	1.0048	1.6725	1.8878

[Table 6]
[New Run] Matrix: size=200 nzero=610 Density=0.025500
[Generate] Diagonal to all elements in the column
[Generate] Time: mul_sub=12, div=8

PARASIM Simulation results
Matrix size=200, Non-zero=610, Density=0.025500
Network: Nodes=609 Links=1019 Tasks=2
Timing: T_{all}=7540 T_{cpa}=220 T_{local}=1
Best Speedup=34.27, CPA =1

Architecture type: 1.

C\P	2	4	8	16	32	64
2	1.0771	2.9580	6.2991	11.5291	19.8945	22.5749
4	1.0732	2.7015	5.2072	10.3288	18.6634	20.9444
8	1.0650	1.9264	4.0978	7.5855	12.9331	17.7412
12	1.0572	1.7333	3.3041	6.3575	11.6179	13.6594
20	1.0420	1.2508	2.3733	4.4301	7.3275	12.0064
32	1.0200	0.9427	1.8000	3.2584	5.1893	7.0270

Architecture type: 2.

C\P	2	4	8	16	32	64
2	1.0768	2.8442	5.1608	5.4598	5.3742	5.3211
4	1.0722	2.3322	2.9732	2.7947	2.7488	2.7191
8	1.0626	1.6492	1.4960	1.4086	1.3950	1.3858
12	1.0528	1.1778	1.0163	0.9495	0.9333	0.9285
20	1.0332	0.7692	0.6033	0.5683	0.5564	0.5564
32	1.0053	0.4636	0.3772	0.3534	0.3485	0.3493

Architecture type: 3.

C\P	2	4	8	16	32	64
2	1.0768	2.8303	5.1608	5.4598	5.3742	5.3211
4	1.0722	2.3861	2.9732	2.7947	2.7488	2.7191
8	1.0626	1.6420	1.4960	1.4086	1.3950	1.3858
12	1.0528	1.1730	1.0163	0.9495	0.9333	0.9285
20	1.0332	0.7684	0.6033	0.5683	0.5564	0.5564
32	1.0053	0.4636	0.3772	0.3534	0.3485	0.3493

[Table 7]
[New Run] Matrix: size=200 nzero=610 Density=0.025500%
[Generate] Diagonal to all elements in the column, Two row elements in one.
[Generate] Time: mul_sub=11, div=6, two_op=20

PARASIM Simulation results
Matrix size=200, Non-zero=610, Density=0.025500
Network: Nodes=449 Links=858 Tasks=3
Timing: T_{all}=6247 T_{cpa}=190 T_{local}=1
Best Speedup=32.88, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.4871	5.4406	11.5240
8	1.9507	4.1012	8.6061
20	1.1844	2.3490	4.8698
40	0.7158	1.3721	2.8254

Architecture type: 2.

C\P	2	4	8	16	32	64
2	1.0690	2.7703	5.7736	8.4191	12.3948	18.8163
4	1.0650	2.5405	4.8314	6.1365	7.5630	15.9362
8	1.0570	1.9775	3.5718	3.5294	3.8777	10.7707
12	1.0492	1.5924	2.5761	2.4643	2.6084	7.2220
20	1.0339	1.1498	1.7134	1.5188	1.5747	3.7953
32	1.0118	0.8245	1.1207	0.9684	1.3300	2.3898

Architecture type: 3.

C\P	2	4	8	16	32	64
2	1.0761	2.9762	5.9495	8.4191	12.3948	18.8163
4	1.0747	2.7814	5.1929	6.1365	7.5630	15.9362
8	1.0717	2.3547	3.3053	3.5294	3.8777	10.7707
12	1.0688	2.0631	2.4345	2.4643	2.6084	7.2220
20	1.0630	1.4657	2.0035	1.5188	1.5747	3.7953
32	1.0543	1.1439	1.0487	0.9684	1.3300	2.3898

Architecture type: 4.

C\P	2	4	8	16	32	64
2	1.0761	2.9691	6.3615	10.5168	19.2215	19.2809
4	1.0747	2.7055	5.8004	9.1868	12.8539	17.1621
8	1.0717	2.4317	4.0644	4.5532	4.8128	12.5694
12	1.0688	1.8743	2.9205	3.1313	3.1188	10.2916
20	1.0630	1.5319	1.9546	2.2733	1.6779	6.9104
32	1.0543	1.1185	1.2947	1.5478	1.5948	3.9613

[Table 8]
[INTERFACE] Matrix: size=200 nzero=610 Density=0.025500%
[Generate] Diagonal to all elements in the column
[Generate] Two row elements in one node
[Generate] Time: mul_sub=11, div=6, two_op=20

PARASIM Simulation results
Matrix size=200, Non-zero=610, Density=0.025500
Network: Nodes=451 Links=860 Tasks=3
Timing: T_{all}=6253 T_{cpa}=225 T_{local}=1
Best Speedup=27.79, CPA On

Architecture type: 2.

C\P	2	4	8	16	32	64
2	1.0735	2.8179	5.5385	8.6727	10.7810	14.3747
4	1.0658	2.5957	4.5181	6.6521	7.3478	8.9201
8	1.0507	1.9990	3.1501	3.5916	4.2221	5.4421
12	1.0361	1.6199	2.6883	1.9329	2.9481	3.9352
20	1.0081	1.1514	1.8252	1.8891	1.8495	2.5533
32	0.9687	0.8008	0.9065	1.1099	1.1823	1.6644

Architecture type: 3.

C\P	2	4	8	16	32	64
2	1.0845	2.8909	5.6081	8.6727	10.7810	14.3747
4	1.0826	2.6306	4.9745	6.6521	7.3478	8.9201
8	1.0788	2.3596	3.0048	3.5916	4.2221	5.4421
12	1.0751	1.9782	2.3767	1.9329	2.9481	3.9352
20	1.0678	1.3979	1.6973	1.8891	1.8495	2.5533
32	1.0570	1.0984	1.0318	1.1099	1.1823	1.6644

Architecture type: 4.

C\P	2	4	8	16	32	64
2	1.0845	2.9152	5.9552	8.5658	10.9702	16.4987
4	1.0826	2.7163	4.8510	7.0022	11.4524	9.6946
8	1.0788	2.3280	3.9278	4.0212	6.6949	4.6595
12	1.0751	2.0016	3.1838	3.6847	2.9762	3.1171
20	1.0678	1.5455	2.1429	1.9914	1.9713	1.7447
32	1.0570	1.0511	1.4853	1.4508	1.8722	1.0909

[Table 9]
[New Run] Matrix: size=200 nzero=610 Density=0.025500%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 5
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=50

PARASIM Simulation results
Matrix size=200, Non-zero=610, Density=0.025500
Network: Nodes=411 Links=820 Tasks=4
Timing: T_{all}=6196 T_{cpa}=224 T_{local}=1
Best Speedup=27.66, CPA On

Architecture type: 2.

C\P	2	4	8	16	32	64
2	1.0751	2.8527	6.0039	9.3313	7.6118	16.7008
4	1.0699	2.4984	4.9097	7.4204	4.1307	13.3247
8	1.0597	2.0422	3.2525	4.9410	2.0695	9.1929
12	1.0496	1.6396	2.2417	3.6772	1.2543	7.2894
20	1.0301	1.2085	1.4069	2.4500	0.7106	4.8031
32	1.0021	0.8378	0.7634	1.6669	0.4283	3.2576

Architecture type: 3.

C\P	2	4	8	16	32	64
2	1.0842	3.0063	5.8563	9.3313	7.6118	16.7008
4	1.0815	2.8487	4.8180	7.4204	4.1307	13.3247
8	1.0763	2.4774	3.6149	4.9410	2.0695	9.1929
12	1.0710	1.9846	2.4924	3.6772	1.2543	7.2894
20	1.0608	1.6085	1.6254	2.4500	0.7106	4.8031
32	1.0457	1.1549	1.0459	1.6669	0.4283	3.2576

Architecture type: 4.

C\P	2	4	8	16	32	64
2	1.0842	2.9947	6.5915	12.6191	14.0499	18.9480
4	1.0815	2.7283	5.1038	7.1137	9.0058	10.3095
8	1.0763	2.4666	3.7552	6.0214	4.2938	5.5321
12	1.0710	2.0169	2.9449	4.1950	2.4384	3.9240
20	1.0608	1.6860	1.8022	2.2771	1.2826	2.5498
32	1.0457	1.2274	1.1697	1.5168	0.7275	1.9815

[Table 10]
[New Run] Matrix: size=200 nzero=610 Density=0.025500%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 5
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=50

PARASIM Simulation results
Matrix size=200, Non-zero=610, Density=0.025500
Network: Nodes=423 Links=832 Tasks=4
Timing: T_{all}=6223 T_{cpa}=237 T_{local}=1
Best Speedup=26.26, CPA On

Architecture type: 2.

C\P	2	4	8	16	32	64
2	1.0766	2.8222	5.4925	7.6356	10.2690	11.9443
4	1.0715	2.5929	4.4010	6.0184	7.5157	9.4145
8	1.0612	1.9557	3.0640	4.0807	4.3917	5.8652
12	1.0512	1.6441	2.2209	3.0386	3.1382	3.9411
20	1.0317	1.1245	1.3540	2.0932	2.0003	2.4862
32	1.0037	0.7796	1.1030	1.3875	1.2956	1.6227

Architecture type: 3.

C\P	2	4	8	16	32	64
2	1.0840	2.9230	5.2737	7.6356	10.2690	11.9443
4	1.0802	2.7621	4.4072	6.0184	7.5157	9.4145
8	1.0727	2.4167	3.1209	4.0807	4.3917	5.8652
12	1.0654	2.0211	2.4675	3.0386	3.1382	3.9411
20	1.0510	1.5256	1.5675	2.0932	2.0003	2.4862
32	1.0301	0.9692	1.0876	1.3875	1.2956	1.6227

Architecture type: 4.

C\P	2	4	8	16	32	64
2	1.0840	2.9395	5.7461	9.0188	12.2742	9.2329
4	1.0802	2.7658	4.8885	7.5157	9.7539	10.5654
8	1.0727	2.4027	3.6692	5.0966	6.7421	6.7863
12	1.0654	2.0270	3.1509	3.3950	4.0780	5.6164
20	1.0510	1.5316	2.1059	2.2929	2.5865	4.1321
32	1.0301	1.1043	1.1829	1.5723	1.6446	2.6038

[Table 11]
[New Run] Matrix: size=200 nzero=610 Density=0.025500%
[Generate] Diagonal to all elements in the column
[Generate] Time: mul_sub=11, div=6, two_op=20

PARASIM Simulation results
Matrix size=200, Non-zero=610, Density=0.025500
Network: Nodes=451 Links=860 Tasks=3
Timing: T_{all}=6253 T_{cpa}=225 T_{local}=1
Best Speedup=27.79, CPA On

Architecture type: 1.

	4	8	16
4	2.6152	5.8330	11.9789
8	2.0685	4.5181	9.1152
20	1.2712	2.6462	5.3081
40	0.7740	1.5621	3.1296

Architecture type: 2.

C\P	2	4	8	16	32	64
2	1.0735	2.8179	5.5385	8.6727	10.7810	14.3747
4	1.0658	2.5957	4.5181	6.6521	7.3478	8.9201
8	1.0507	1.9990	3.1501	3.5916	4.2221	5.4421
12	1.0361	1.6199	2.6883	1.9329	2.9481	3.9352
20	1.0081	1.1514	1.8252	1.8891	1.8495	2.5533
32	0.9687	0.8008	0.9065	1.1099	1.1823	1.6644

Architecture type: 3.

C\P	2	4	8	16	32	64
2	1.0845	2.8909	5.6081	8.6727	10.7810	14.3747
4	1.0826	2.6306	4.9745	6.6521	7.3478	8.9201
8	1.0788	2.3596	3.0048	3.5916	4.2221	5.4421
12	1.0751	1.9782	2.3767	1.9329	2.9481	3.9352
20	1.0678	1.3979	1.6973	1.8891	1.8495	2.5533
32	1.0570	1.0984	1.0318	1.1099	1.1823	1.6644

Architecture type: 4.

C\P	2	4	8	16	32	64
2	1.0845	2.9152	5.9552	8.5658	10.9702	16.4987
4	1.0826	2.7163	4.8510	7.0022	11.4524	9.6946
8	1.0788	2.3280	3.9278	4.0212	6.6949	4.6595
12	1.0751	2.0016	3.1838	3.6847	2.9762	3.1171
20	1.0678	1.5455	2.1429	1.9914	1.9713	1.7447
32	1.0570	1.0511	1.4853	1.4508	1.8722	1.0909

[Table 12]
[INTERFACE] Matrix: size=200 nzero=610 Density=0.025500%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 10
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=100

PARASIM Simulation results
Matrix size=200, Non-zero=610, Density=0.025500
Network: Nodes=449 Links=858 Tasks=4
Timing: T_{all}=6247 T_{cpa}=190 T_{local}=1
Best Speedup=32.88, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.5825	5.7896	11.6985
8	2.0665	4.4526	9.2139
20	1.2851	2.6303	5.5234
40	0.7867	1.5579	3.3001

Architecture type: 2.

C\P	4	8	16
4	2.5405	4.8314	6.1365
8	1.9775	3.5718	3.5294
16	1.3440	2.0782	1.8936

Architecture type: 3.

C\P	4	8	16
4	2.7814	5.1929	6.1365
8	2.3547	3.3053	3.5294
16	1.6614	2.2808	1.8936

Architecture type: 4.

C\P	4	8	16
4	2.7055	5.8004	9.1868
8	2.4317	4.0644	4.5532
16	1.7078	2.2184	2.7399

Model 300

[Table 13]
[New Run] Matrix: size=300 nzero=1203 Density=0.023400%
[Generate] Diagonal to all elements in the column
[Generate] Two row elements in one node
[Generate] Time: mul_sub=11, div=6, two_op=20

PARASIM Simulation results
Matrix size=300, Non-zero=1203, Density=0.023400
Network: Nodes=817 Links=1719 Tasks=3
Timing: T_{all}=12681 T_{cpa}=470 T_{local}=1
Best Speedup=26.98, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.5286	5.7354	11.9519
8	2.0005	4.4793	9.1296
20	1.2299	2.6529	5.3282
40	0.7483	1.5772	3.1234

Architecture type: 2.

C\P	4	8	16
4	2.5433	4.3221	7.3301
8	1.8921	2.6652	3.6273
16	1.3065	1.4993	1.8074

Architecture type: 3.

C\P	4	8	16
4	2.7502	4.8848	7.3301
8	2.4012	3.1428	3.6273
16	1.6752	2.0093	1.8074

Architecture type: 4.

C\P	4	8	16
4	2.7271	5.0361	8.9052
8	2.4049	3.8184	6.1588
16	1.7060	2.4982	2.8860

[Table 14]
[INTERFACE] Matrix: size=2352 nzero=16465 Density=0.005528%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 5
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=50

PARASIM Simulation results
Matrix size=2352, Non-zero=16465, Density=0.005528
Network: Nodes=6303 Links=20415 Tasks=4
Timing: T_{all}=176443 T_{cpa}=7732 T_{local}=1
Best Speedup=22.82, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.6336	5.8423	12.2700
8	2.1322	4.5552	9.5498
20	1.3552	2.7401	5.6961
40	0.8427	1.6458	3.4012

Architecture type: 2.

C\P	4	8	16
4	2.5748	4.7665	4.4805
8	1.9871	3.1785	2.7548
16	1.3018	1.6814	1.4488

Architecture type: 3.

C\P	4	8	16
4	2.9348	6.0794	4.5921
8	2.7321	5.1061	2.5409
16	2.2962	3.5856	1.3751

Architecture type: 4.

C\P	4	8	16
4	2.9403	6.2023	5.1907
8	2.7246	5.1498	4.1730
16	2.3308	3.9584	3.6955

[Table 15]
[INTERFACE] Matrix: size=2352 nzero=16465 Density=0.005528%
[Generate] Time: mul_sub=11, div=6, two_op=20

PARASIM Simulation results
Matrix size=2352, Non-zero=16465, Density=0.005528
Network: Nodes=9852 Links=23964 Tasks=3
Timing: T_{all}=180094 T_{cpa}=4100 T_{local}=1
Best Speedup=43.93, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.4899	5.5068	11.4615
8	1.9576	4.1728	8.6054
20	1.1896	2.4049	4.9020
40	0.7185	1.4091	2.8514

Architecture type: 2.

C\P	4	8	16
4	2.4658	4.6388	4.3908
8	1.7714	2.7090	2.6454
16	1.1243	1.4544	1.4726

Architecture type: 3.

C\P	4	8	16
4	2.8630	5.5242	4.5312
8	2.5859	4.5140	2.7217
16	2.0034	3.0598	1.4265

Architecture type: 4.

C\P	4	8	16
4	2.8667	5.5724	7.1671
8	2.5808	4.5438	5.0746
16	2.0284	3.2369	3.0380

[Table 16]
[INTERFACE] Matrix: size=2352 nzero=16556 Density=0.005560%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 5
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=50

PARASIM Simulation results
Matrix size=2352, Non-zero=16556, Density=0.005560
Network: Nodes=6133 Links=20336 Tasks=4
Timing: T_{all}=176981 T_{cpa}=6275 T_{local}=1
Best Speedup=28.20, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.6089	5.8082	12.1212
8	2.1127	4.5379	9.4718
20	1.3422	2.7357	5.6562
40	0.8346	1.6461	3.3769

Architecture type: 2.

C\P	4	8	16
4	2.5313	4.9413	4.7255
8	2.0318	3.1342	2.8846
16	1.3296	1.7232	1.6777

Architecture type: 3.

C\P	4	8	16
4	2.9085	6.0866	4.7255
8	2.6844	4.9334	2.8841
16	2.3090	3.5448	1.6777

Architecture type: 4.

C\P	4	8	16
4	2.9098	6.0950	5.4994
8	2.6873	5.2627	4.8460
16	2.3058	4.0250	3.7328

[Table 17]
[INTERFACE] Matrix: size=2352 nzero=16556 Density=0.005560%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 10
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=100

PARASIM Simulation results
Matrix size=2352, Non-zero=16556, Density=0.005560
Network: Nodes=6223 Links=20426 Tasks=4
Timing: T_{all}=177285 T_{cpa}=20545 T_{local}=1
Best Speedup=8.63, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.8493	6.0439	6.5566
8	2.3090	4.8314	5.3269
20	1.4657	2.9640	3.3978
40	0.9092	1.7996	2.1175

Architecture type: 2.

C\P	4	8	16
4	2.7403	4.5606	4.2586
8	2.0497	3.0612	2.7796
16	1.3277	1.8162	1.4453

Architecture type: 3.

C\P	4	8	16
4	3.1702	5.8924	4.2258
8	2.9241	4.9475	2.7523
16	2.5223	3.6186	1.4569

Architecture type: 4.

C\P	4	8	16
4	3.1711	6.0125	3.3394
8	2.9176	5.2811	3.0033
16	2.5308	4.0094	2.9213

[Table 18]
[INTERFACE] Matrix: size=2352 nzero=16465 Density=0.005528%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 10
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=100

PARASIM Simulation results
Matrix size=2352, Non-zero=16465, Density=0.005528
Network: Nodes=6916 Links=21028 Tasks=4
Timing: T_{all}=177158 T_{cpa}=11096 T_{local}=1
Best Speedup=15.97, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.6579	5.8650	11.6413
8	2.1497	4.6050	9.2179
20	1.3580	2.7722	5.6330
40	0.8403	1.6634	3.3937

Architecture type: 2.

C\P	4	8	16
4	2.5724	4.3722	4.1052
8	1.8614	2.5823	2.9003
16	1.2230	1.4255	1.4578

Architecture type: 3.

C\P	4	8	16
4	2.9451	5.8839	4.7137
8	2.7126	4.7399	2.6590
16	2.2682	3.5549	1.6246

Architecture type: 4.

C\P	4	8	16
4	2.9743	5.8127	6.8151
8	2.7342	4.9063	3.7554
16	2.2737	3.6321	2.9321

Model 2352

[Table 19]
[New Run] Matrix: size=2352 nzero=22642 Density=0.007761
[Generate] Diagonal to all elements in the column
[Generate] Time: mul_sub=8, div=4

PARASIM Simulation results
Matrix size=2352, Non-zero=22642, Density=0.007761
Network: Nodes=22641 Links=42931 Tasks=2
Timing: T_{all}=214660 T_{cpa}=3412 T_{local}=1
Best Speedup=62.91, CPA On

Architecture type: 2.

C\P	2	4	8	16	32	64
2	1.0260	1.6202	4.0516	5.4958	6.9474	8.1397
4	1.0165	1.4980	3.1327	3.6884	4.6539	5.1015
8	0.9980	1.2734	1.9728	2.2195	2.7048	2.8696
12	0.9801	1.1316	1.4255	1.6039	1.8574	1.9390
20	0.9463	0.8457	0.9310	0.9942	1.1260	1.1673
32	0.8997	0.5462	0.6048	0.6382	0.7396	0.7255

Architecture type: 3.

C\P	2	4	8	16	32	64
2	1.0430	1.7872	4.8800	7.0974	6.9474	8.1397
4	1.0419	1.7412	4.2710	5.2798	4.6539	5.1015
8	1.0397	1.6380	3.3779	3.0030	2.7048	2.8696
12	1.0375	1.4769	2.6584	2.0647	1.8574	1.9390
20	1.0331	1.1683	1.4292	1.2968	1.1260	1.1673
32	1.0266	0.9294	1.1532	0.8440	0.7396	0.7255

Architecture type: 4.

C\P	2	4	8	16	32	64
2	1.0430	1.7872	4.9215	7.1503	8.6752	10.7631
4	1.0419	1.7413	4.2863	5.6900	4.4254	6.1351
8	1.0397	1.6382	3.2542	2.9107	2.9724	4.4767
12	1.0375	1.4773	2.6678	2.3730	2.2896	2.6544
20	1.0331	1.1683	1.7202	1.3108	1.5677	1.8637
32	1.0266	0.9291	1.0346	0.9553	0.8663	1.3917

[Table 20]
[New Run] Matrix: size=2352 nzero=22642 Density=0.007761
[Generate] Diagonal to all elements in the column
[Generate] Time: mul_sub=8, div=4

PARASIM Simulation results
Matrix size=2352, Non-zero=22642, Density=0.007761%
Network: Nodes=22641 Links=42931 Tasks=2
Timing: T_{all}=214660 T_{cpa}=18572 T_{local}=1
Best Speedup=11.56, CPA On

Architecture type: 2.

C\P	2	4	8	16	32	64
2	1.0703	1.6367	3.6008	4.1683	4.9682	6.3316
4	0.9718	1.4536	2.7698	3.1447	3.9001	5.1429
8	0.8196	1.1787	1.8466	2.0361	2.5038	2.8597
12	0.7085	0.9425	1.3349	1.4891	1.4952	2.6563
20	0.5574	0.6550	0.8797	0.9810	1.1558	1.5881
32	0.4222	0.4463	0.5790	0.6355	0.7317	1.0538

Architecture type: 3.

C\P	2	4	8	16	32	64
2	1.1848	1.9089	3.9042	4.1683	4.9682	6.3316
4	1.1715	1.8305	3.3528	3.1447	3.9001	5.1429
8	1.1456	1.6756	2.6540	2.0361	2.5038	2.8597
12	1.1209	1.5111	2.0469	1.4891	1.4952	2.6563
20	1.0746	1.2016	1.4828	0.9810	1.1558	1.5881
32	1.0119	0.9088	0.9574	0.6355	0.7317	1.0538

Architecture type: 4.

C\P	2	4	8	16	32	64
2	1.1848	1.9089	3.8977	5.2070	5.5646	7.7970
4	1.1715	1.8305	3.4931	3.9306	4.4803	6.2267
8	1.1456	1.6756	2.6439	2.7649	3.1279	3.2932
12	1.1209	1.5111	2.0891	2.0370	2.1962	3.5468
20	1.0746	1.2016	1.4823	1.3343	1.3811	2.1470
32	1.0119	0.9088	1.0000	0.9374	1.0408	1.4404

[Table 21]
[New Run] Matrix: size=2352 nzero=22642 Density=0.007761%
[Generate] Diagonal to all elements in the column
[Generate] New method
[Generate] Time: mul_sub=11, div=6

PARASIM Simulation results
Matrix size=2352, Non-zero=22642, Density=0.007761
Network: Nodes=22641 Links=42931 Tasks=2
Timing: T_{all}=280234 T_{cpa}=26054 T_{local}=1
Best Speedup=10.76, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.5296	5.6073	7.9328
8	1.8921	4.0915	6.4043
20	1.0768	2.2544	3.7514
40	0.6267	1.2877	2.1819

[Table 22] [New Run] Matrix: size=2352 nzero=22642 Density=0.007761%
[Generate] Diagonal to all elements in the column
[Generate] Time: mul_sub=11, div=6, two_op=20

PARASIM Simulation results
Matrix size=2352, Non-zero=22642, Density=0.007761
Network: Nodes=13124 Links=33413 Tasks=3
Timing: T_{all}=251680 T_{cpa}=27167 T_{local}=1
Best Speedup=9.26, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.6555	5.6749	7.1563
8	2.0271	4.2884	6.0558
20	1.1840	2.4706	3.9712
40	0.6987	1.4472	2.3892

Architecture type: 2.

C\P	2	4	8	16	32	64
2	1.0976	2.9221	5.5070	4.8377	5.7888	7.0374
4	1.0096	2.5403	4.2755	3.7513	4.7568	5.8381
8	0.8702	1.7815	2.5151	2.5552	3.5909	4.4897
12	0.7646	1.3724	1.7884	1.7134	2.9771	3.3056
20	0.6146	0.9402	1.1130	1.2930	1.9349	2.1832
32	0.4746	0.6380	0.7310	0.8709	1.3280	1.3958

Architecture type: 3.

C\P	2	4	8	16	32	64
2	1.1968	3.2399	5.8235	4.8377	5.7888	7.0374
4	1.1858	3.0917	5.2893	3.7513	4.7568	5.8381
8	1.1636	2.8162	4.4211	2.5552	3.5909	4.4897
12	1.1416	2.4757	3.6649	1.7134	2.9771	3.3056
20	1.1006	1.9452	2.7092	1.2930	1.9349	2.1832
32	1.0434	1.4321	1.7771	0.8709	1.3280	1.3958

Architecture type: 4.

C\P	2	4	8	16	32	64
2	1.1968	3.2399	5.8464	5.8094	6.3737	7.1689
4	1.1858	3.0917	5.2790	5.4178	5.6376	6.6536
8	1.1636	2.8162	4.5081	4.3052	4.6673	5.9962
12	1.1416	2.4757	3.7201	3.5088	4.3010	5.1797
20	1.1006	1.9452	2.7025	2.4827	2.6560	4.0831
32	1.0434	1.4321	1.9295	1.7840	2.3977	2.2272

[Table 23]
[New Run] Matrix: size=2352 nzero=22642 Density=0.007761%
[Generate] Diagonal to all elements in the column
[Generate] Two row elements in one node
[Generate] Time: mul_sub=11, div=6, two_op=20

PARASIM Simulation results
Matrix size=2352, Non-zero=22642, Density=0.007761
Network: Nodes=13231 Links=33520 Tasks=3
Timing: T_{all}=252001 T_{cpa}=8207 T_{local}=1
Best Speedup=30.71, CPA On

Architecture type: 2.

C\P	2	4	8	16	32	64
2	1.0377	2.7776	5.8807	6.1739	7.0866	7.8402
4	1.0164	2.4421	4.5442	4.1813	5.2995	6.6173
8	0.9763	1.7249	2.6326	2.7271	3.2571	3.6205
12	0.9392	1.3376	1.8732	1.6570	2.5064	2.8836
20	0.8729	0.9115	1.1402	0.9661	1.3498	2.1532
32	0.7893	0.6274	0.7600	0.6647	0.8939	1.3425

Architecture type: 3.

C\P	2	4	8	16	32	64
2	1.0635	3.0596	6.3882	6.5075	7.0866	7.8402
4	1.0609	2.9215	5.6957	4.6745	5.2995	6.6173
8	1.0558	2.6414	4.5531	2.9625	3.2571	3.6205
12	1.0508	2.3214	3.7685	2.0754	2.5064	2.8836
20	1.0408	1.8037	2.5952	1.3278	1.3498	2.1532
32	1.0262	1.3306	1.8934	0.8957	0.8939	1.3425

Architecture type: 4.

C\P	2	4	8	16	32	64
2	1.0635	3.0599	6.3979	8.2778	8.2688	10.1478
4	1.0609	2.9201	5.7118	6.9648	7.6708	9.6486
8	1.0558	2.6337	4.6420	5.1705	5.9014	7.2957
12	1.0508	2.3313	3.8966	3.9297	4.5338	5.1441
20	1.0408	1.8040	2.8133	2.8190	2.8618	3.8092
32	1.0262	1.3228	1.9315	1.8492	2.1896	2.3488

[Table 24]
[New Run] Matrix: size=2352 nzero=22642 Density=0.007761%
[Generate] Diagonal to all elements in the column
[Generate] Two row elements in one node
[Generate] Time: mul_sub=11, div=6, two_op=20

PARASIM Simulation results
Matrix size=2352, Non-zero=22642, Density=0.007761
Network: Nodes=12995 Links=33284 Tasks=3
Timing: T_{all}=251293 T_{cpa}=6333 T_{local}=1
Best Speedup=39.68, CPA On

Architecture type: 2.

C\P	2	4	8	16	32	64
2	1.0324	2.7185	5.7537	8.8409	8.6254	11.4651
4	1.0190	2.4371	4.8151	5.7523	4.6516	7.5577
8	0.9932	1.8405	3.2149	3.1692	3.0858	3.5653
12	0.9686	1.4647	2.2940	2.4283	2.2403	2.7263
20	0.9230	0.9900	1.4820	1.4788	1.3159	1.4579
32	0.8621	0.6855	0.9364	0.9027	0.8027	0.9024

Architecture type: 3.

C\P	2	4	8	16	32	64
2	1.0514	3.0597	6.4545	12.2331	8.6254	11.4651
4	1.0498	2.9291	5.7045	8.5619	4.6516	7.5577
8	1.0468	2.6092	4.5970	4.5018	3.0858	3.5653
12	1.0438	2.3204	3.6769	3.1425	2.2403	2.7263
20	1.0378	1.8305	2.6930	1.7422	1.3159	1.4579
32	1.0289	1.3660	1.7777	1.4543	0.8027	0.9024

Architecture type: 4.

C\P	4	8	16	32	6	
2	1.0514	3.0598	6.4831	12.7839	14.8853	17.5508
4	1.0498	2.9291	5.7991	10.6471	9.4397	12.7140
8	1.0468	2.6092	4.6871	6.6453	6.0959	6.5068
12	1.0438	2.3212	3.8603	5.2315	5.4451	5.2827
20	1.0378	1.8305	2.8076	3.3042	3.0231	2.4850
32	1.0289	1.3662	2.0180	1.9754	1.7646	2.3590

[Table 25]
[New Run] Matrix: size=2352 nzero=22642 Density=0.007761%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 5
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=50

PARASIM Simulation results
Matrix size=2352, Non-zero=22642, Density=0.007761
Network: Nodes=8474 Links=28763 Tasks=4
Timing: T_{all}=247030 T_{cpa}=54167 T_{local}=1
Best Speedup=4.56, CPA On

Architecture type: 2.

C\P	2	4	8	16	32	64
2	1.2825	3.3722	3.5382	3.8374	4.0057	4.2196
4	1.1851	3.0003	3.3043	3.7971	3.9114	3.9850
8	1.0286	2.2583	2.6363	3.2109	3.3864	3.8226
12	0.9081	1.7346	1.9885	2.2777	2.9045	3.0425
20	0.7352	1.1998	1.2398	1.6502	1.9725	2.3032
32	0.5718	0.8200	0.7874	1.0808	1.3443	1.5814

Architecture type: 3.

C\P	2	4	8	16	32	64
2	1.3936	3.6636	4.1174	3.8374	4.0057	4.2196
4	1.3773	3.5664	4.0467	3.7971	3.9114	3.9850
8	1.3452	3.2765	3.8214	3.2109	3.3864	3.8226
12	1.3140	2.9594	3.5956	2.2777	2.9045	3.0425
20	1.2561	2.3889	3.0576	1.6502	1.9725	2.3032
32	1.1776	1.8557	2.3610	1.0808	1.3443	1.5814

Architecture type: 4.

C\P	2	4	8	16	32	64
2	1.3936	3.6636	4.1462	3.9865	4.2132	4.1606
4	1.3773	3.5664	3.9974	3.9616	4.2211	4.1565
8	1.3452	3.2765	3.8798	3.8232	4.0713	4.1725
12	1.3140	2.9594	3.6249	3.8609	4.0276	4.0792
20	1.2561	2.3889	3.3469	3.1594	3.7577	4.0241
32	1.1776	1.8557	2.4278	2.6306	3.3294	3.3248

[Table 26]
[INTERFACE] Matrix: size=2352 nzero=22642 Density=0.007761%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 5
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=50

PARASIM Simulation results
Matrix size=2352, Non-zero=22642, Density=0.007761
Network: Nodes=8540 Links=28829 Tasks=4
Timing: T_{all}=246913 T_{cpa}=15399 T_{local}=1
Best Speedup=16.03, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.6587	5.8152	11.1513
8	2.1343	4.5373	8.7434
20	1.3387	2.7201	5.2956
40	0.8250	1.6303	3.1841

Architecture type: 2.

C\P	4	8	16
4	2.5810	4.0386	3.7910
8	1.9557	2.5273	2.5868
16	1.2592	1.3745	1.6345

Architecture type: 3.

C\P	4	8	16
4	3.0412	5.5389	4.1840
8	2.8263	4.4490	2.9874
16	2.3169	3.2422	1.8594

Architecture type: 4.

C\P	4	8	16
4	3.0416	5.3172	4.7310
8	2.8361	4.7203	4.4885
16	2.3247	3.4181	3.6416

[Table 27]
[INTERFACE] Matrix: size=2352 nzero=22642 Density=0.007761%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 5
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=50

PARASIM Simulation results
Matrix size=2352, Non-zero=22642, Density=0.007761
Network: Nodes=7453 Links=27742 Tasks=4
Timing: T_{all}=245488 T_{cpa}=11246 T_{local}=1
Best Speedup=21.83, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.6108	5.8721	12.2200
8	2.0968	4.5797	9.3730
20	1.3166	2.7491	5.5100
40	0.8122	1.6498	3.2648

Architecture type: 2.

C\P	4	8	16
4	2.5007	5.1561	5.1395
8	2.0637	3.6012	3.2487
16	1.4051	2.0916	1.8817

Architecture type: 3.

C\P	4	8	16
4	3.0137	6.4004	9.1334
8	2.8100	5.4820	5.5198
16	2.4442	3.8630	3.5134

Architecture type: 4.

C\P	4	8	16
4	3.0123	6.4397	12.0686
8	2.8092	5.7174	8.4239
16	2.4454	4.3907	5.1935

[Table 28]
[INTERFACE] Matrix: size=2352 nzero=22642 Density=0.007761%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 10
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=100

PARASIM Simulation results
Matrix size=2352, Non-zero=22642, Density=0.007761
Network: Nodes=7455 Links=27744 Tasks=4
Timing: T_{all}=245753 T_{cpa}=11200 T_{local}=1
Best Speedup=21.94, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.6294	5.8453	12.2834
8	2.1254	4.5472	9.4970
20	1.3485	2.7242	5.6222
40	0.8379	1.6328	3.3447

Architecture type: 2.

C\P	4	8	16
4	2.5092	4.9596	4.2884
8	2.0458	2.9643	2.7917
16	1.3680	1.6617	1.6847

Architecture type: 3.

C\P	4	8	16
4	2.9640	6.1264	7.9033
8	2.7590	5.3982	5.5413
16	2.4149	3.9990	3.1293

Architecture type: 4.

C\P	4	8	16
4	2.9640	6.2471	9.4176
8	2.7575	5.6602	7.5833
16	2.4293	4.3285	5.0351

[Table 29]
[INTERFACE] Matrix: size=2352 nzero=22642 Density=0.007761%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 10
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=100

PARASIM Simulation results
Matrix size=2352, Non-zero=22642, Density=0.007761
Network: Nodes=7444 Links=27733 Tasks=4
Timing: T_{all}=246000 T_{cpa}=95707 T_{local}=1
Best Speedup=2.57, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.1446	2.1602	2.1770
8	1.9616	2.0060	2.0338
20	1.5056	1.5536	1.6009
40	1.0651	1.1100	1.1498

Architecture type: 2.

C\P	4	8	16
4	2.1968	2.1081	2.2878
8	2.0469	1.9132	2.1299
16	1.7745	1.5740	1.6809

Architecture type: 3.

C\P	4	8	16
4	2.3079	2.1964	2.2878
8	2.2211	2.2080	2.1299
16	2.0806	2.1701	1.6809

Architecture type: 4.

C\P	4	8	16
4	2.3079	2.2247	2.3305
8	2.2211	2.1929	2.3389
16	2.0806	2.1690	2.2138

[Table 30]
[INTERFACE] Matrix: size=2352 nzero=22642 Density=0.007761%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 10
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=100

PARASIM Simulation results
Matrix size=2352, Non-zero=22642, Density=0.007761
Network: Nodes=8531 Links=28820 Tasks=4
Timing: T_{all}=247301 T_{cpa}=20107 T_{local}=1
Best Speedup=12.30, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.6998	6.0283	9.2016
8	2.1704	4.7235	7.3586
20	1.3615	2.8486	4.5356
40	0.8388	1.7104	2.7554

Architecture type: 2.

C\P	4	8	16
4	2.5945	3.3844	2.7844
8	1.9598	2.4230	2.4226
16	1.2821	1.5005	1.8331

Architecture type: 3.

C\P	4	8	16
4	3.0144	4.6920	2.7330
8	2.7644	4.0433	2.5221
16	2.3500	3.1095	1.8672

Architecture type: 4.

C\P	4	8	16
4	3.0095	4.8937	3.9235
8	2.7633	4.0295	3.3255
16	2.3395	3.3187	2.7237

[Table 31]
[INTERFACE] Matrix: size=3516 nzero=59741 Density=0.009381%
[Generate] Diagonal to all elements in the column
[Generate] Two row elements in one node
[Generate] Time: mul_sub=11, div=6, two_op=20

PARASIM Simulation results
Matrix size=3516, Non-zero=59741, Density=0.009381
Network: Nodes=32384 Links=88608 Tasks=3
Timing: T_{all}=673466 T_{cpa}=12335 T_{local}=1
Best Speedup=54.60, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.4184	5.3738	11.2873
8	1.8778	4.0229	8.3477
20	1.1230	2.2864	4.6774
40	0.6722	1.3293	2.6978

Architecture type: 2.

C\P	4	8	16
4	2.3396	4.6846	4.5836
8	1.6460	2.6318	2.6109
16	1.0264	1.4458	1.4738

Architecture type: 3.

C\P	4	8	16
4	2.9108	5.8390	6.9743
8	2.6573	4.8113	4.3659
16	2.0972	3.3382	2.3425

Architecture type: 4.

C\P	4	8	16
4	2.9099	5.8398	8.3307
8	2.6622	4.8434	5.2609
16	2.1144	3.5271	3.1040

[Table 32]
[INTERFACE] Matrix: size=3516 nzero=59376 Density=0.009322%
[Generate] Diagonal to all elements in the column
[Generate] Two row elements in one node
[Generate] Time: mul_sub=11, div=6, two_op=20

PARASIM Simulation results
Matrix size=3516, Non-zero=59376, Density=0.009322
Network: Nodes=32247 Links=88106 Tasks=3
Timing: T_{all}=669405 T_{cpa}=17109 T_{local}=1
Best Speedup=39.13, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.4345	5.4079	11.3188
8	1.8863	4.0454	8.3676
20	1.1239	2.2989	4.6879
40	0.6715	1.3357	2.7033

Architecture type: 2.

C\P	4	8	16
4	2.3642	4.5392	4.3507
8	1.6550	2.6498	2.7964
16	1.0278	1.4136	1.3977

Architecture type: 3.

C\P	4	8	16
4	2.9303	5.8222	7.1413
8	2.6843	4.8139	4.6229
16	2.1226	3.4393	2.4675

Architecture type: 4.

C\P	4	8	16
4	2.9319	5.8684	7.6933
8	2.6865	4.8285	5.1754
16	2.1193	3.4875	3.1354

[Table 33]
[INTERFACE] Matrix: size=3516 nzero=59741 Density=0.009381%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 5
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=50

PARASIM Simulation results
Matrix size=3516, Non-zero=59741, Density=0.009381
Network: Nodes=16941 Links=73165 Tasks=4
Timing: T_{all}=657653 T_{cpa}=23307 T_{local}=1
Best Speedup=28.22, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.5412	5.7041	11.8453
8	2.0248	4.4088	9.1126
20	1.2566	2.6200	5.3698
40	0.7694	1.5627	3.1775

Architecture type: 2.

C\P	4	8	16
4	2.4793	4.5999	3.9806
8	1.7830	2.7386	2.6643
16	1.1349	1.5661	1.3721

Architecture type: 3.

C\P	4	8	16
4	2.9956	6.3450	6.6725
8	2.8554	5.5724	5.4655
16	2.5561	4.5248	3.6235

Architecture type: 4.

C\P	4	8	16
4	2.9958	6.3945	6.9667
8	2.8553	5.5619	6.7982
16	2.5614	4.6022	4.7129

[Table 34]
[INTERFACE] Matrix: size=3516 nzero=59376 Density=0.009322%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 5
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=50

PARASIM Simulation results
Matrix size=3516, Non-zero=59376, Density=0.009322
Network: Nodes=16881 Links=72740 Tasks=4
Timing: T_{all}=653697 T_{cpa}=32246 T_{local}=1
Best Speedup=20.27, CPA On

Architecture type:1.

C\P	4	8	16
4	2.5762	5.7907	11.7709
8	2.0485	4.4688	9.1019
20	1.2674	2.6493	5.3842
40	0.7747	1.5778	3.2017

Architecture type: 2.

C\P	4	8	16
4	2.5045	4.2287	4.2098
8	1.8220	2.5021	2.6790
16	1.1723	1.3526	1.4840

Architecture type: 3.

C\P	4	8	16
4	3.0406	6.4046	6.9531
8	2.8823	5.6332	5.3651
16	2.5862	4.5938	3.8102

Architecture type: 4.

C\P	4	8	16
4	3.0399	6.4362	7.5592
8	2.8882	5.7448	6.0306
16	2.5802	4.5995	4.7445

[Table 35]
[INTERFACE] Matrix: size=3516 nzero=59741 Density=0.009381%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 10
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=100

PARASIM Simulation results
Matrix size=3516, Non-zero=59741, Density=0.009381
Network: Nodes=14412 Links=70636 Tasks=4
Timing: T_{all}=655494 T_{cpa}=42513 T_{local}=1
Best Speedup=15.42, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.6429	5.9545	10.8098
8	2.1220	4.6639	8.5299
20	1.3311	2.8187	5.2013
40	0.8207	1.6977	3.1423

Architecture type: 2.

C\P	4	8	16
4	2.5661	4.1291	3.4067
8	1.8861	2.4792	2.2685
16	1.2115	1.4477	1.3913

Architecture type: 3.

C\P	4	8	16
4	3.0605	6.0799	4.8426
8	2.9248	5.5293	4.4261
16	2.6422	4.4113	3.3045

Architecture type: 4.

C\P	4	8	16
4	3.0525	6.0738	5.1181
8	2.9142	5.4951	4.8192
16	2.6451	4.3583	4.3196

[Table 36]
[INTERFACE] Matrix: size=3516 nzero=59376 Density=0.009322%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 10
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=100

PARASIM Simulation results
Matrix size=3516, Non-zero=59376, Density=0.009322
Network: Nodes=14959 Links=70818 Tasks=4
Timing: T_{all}=652117 T_{cpa}=58663 T_{local}=1
Best Speedup=11.12, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.7267	6.1184	8.4265
8	2.1910	4.7852	6.8078
20	1.3750	2.8842	4.2838
40	0.8476	1.7336	2.6363

Architecture type: 2.

C\P	4	8	16
4	2.6154	3.6035	3.7175
8	1.9487	2.1485	2.5336
16	1.2447	1.1718	1.6935

Architecture type: 3.

C\P	4	8	16
4	3.1088	5.2900	4.7172
8	2.9539	4.7853	4.0993
16	2.6679	3.8775	3.5303

Architecture type: 4.

C\P	4	8	16
4	3.1104	5.4197	4.8734
8	2.9935	4.8122	4.7104
16	2.6736	4.0020	4.0666

Model 3516

[Table 37]
[New Run] Matrix: size=3516 nzero=81390 Density=0.012883
[Generate] Diagonal to all elements in the column
[Generate] Time: mul_sub=8, div=4

PARASIM Simulation results
Matrix size=3516, Non-zero=81390, Density=0.012883
Network: Nodes=81389 Links=159263 Tasks=5
Timing: T_{all}=796320 T_{cpa}=21200 T_{local}=1
Best Speedup=37.56, CPA On

Architecture type: 2.

C\P	2	4	8	16	32	64
2	1.0049	1.5737	4.0018	5.7037	5.6864	6.5106
4	0.9449	1.4060	2.9463	3.7744	3.6186	4.5567
8	0.8439	1.1523	1.8645	2.2515	2.1080	2.8643
12	0.7623	0.9472	1.3490	1.5879	1.4154	1.9568
20	0.6389	0.6678	0.8627	0.9977	0.9719	1.2678
32	0.5140	0.4552	0.5613	0.6420	0.5764	0.8794

Architecture type: 3.

C\P	2	4	8	16	32	64
2	1.0792	1.8188	4.4328	6.1495	5.7806	6.5106
4	1.0757	1.7804	3.7990	4.1816	3.9214	4.5567
8	1.0688	1.6297	2.7272	2.4269	2.2029	2.8643
12	1.0619	1.4418	2.0506	1.6854	1.5016	1.9568
20	1.0484	1.1445	1.2482	0.9614	0.9676	1.2678
32	1.0288	0.8798	0.8850	0.6880	0.6008	0.8794

Architecture type: 4.

C\P	2	4	8	16	32	64
2	1.0792	1.8188	4.4412	6.2795	6.3615	7.8057
4	1.0757	1.7804	3.8146	4.0942	3.9392	4.7754
8	1.0688	1.6297	2.7285	2.4301	2.2411	2.5890
12	1.0619	1.4418	1.9538	1.8315	1.5914	1.8899
20	1.0484	1.1445	1.3645	1.0582	0.9673	1.1843
32	1.0288	0.8798	0.7943	0.7043	0.6450	0.8178

[Table 38]
[New Run] Matrix: size=3516 nzero=81390 Density=0.012883%
[Generate] Diagonal to all elements in the column
[Generate] New method
[Generate] Time: mul_sub=11, div=6

PARASIM Simulation results
Matrix size=3516, Non-zero=81390, Density=0.012883
Network: Nodes=81389 Links=159263 Tasks=2
Timing: T_{all}=1036974 T_{cpa}=29751 T_{local}=1
Best Speedup=34.86, CPA On

Architecture type: 2.

C\P	4	8	16
4	2.2461	4.4942	4.4723
8	1.5342	2.5410	2.5276
16	0.9379	1.3606	1.3643

Architecture type: 3.

C\P	4	8	16
4	2.8373	5.2457	5.1758
8	2.3386	4.0079	2.8329
16	1.6236	2.4210	1.4788

Architecture type: 4.

C\P	4	8	16
4	2.8373	5.1832	5.4929
8	2.3386	4.0686	2.9570
16	1.6236	2.4325	1.6718

[Table 39]
[New Run] Matrix: size=3516 nzero=81390 Density=0.012883%
[Generate] Diagonal to all elements in the column
[Generate] Two row elements in one node
[Generate] Time: mul_sub=11, div=6, two_op=20

PARASIM Simulation results
Matrix size=3516, Non-zero=81390, Density=0.012883
Network: Nodes=43190 Links=121063 Tasks=3
Timing: T_{all}=922374 T_{cpa}=3585 T_{local}=1
Best Speedup=257.29, CPA On

Architecture type: 2.

C\P	2	4	8	16	32	64
2	1.0049	2.6774	5.7322	8.7221	10.8595	13.7956
4	0.9983	2.3532	4.6533	5.6144	6.0264	8.7231
8	0.9853	1.6749	2.7323	3.0788	3.1228	4.7122
12	0.9726	1.2956	1.9107	2.2378	2.2414	3.5023
20	0.9482	0.8918	1.2247	1.3456	1.2856	1.8799
32	0.9138	0.6115	0.7913	0.8823	0.7954	1.3355

Architecture type: 3.

C\P	2	4	8	16	32	64
2	1.0156	2.9875	6.4320	12.6595	14.6327	18.5951
4	1.0154	2.8822	5.7740	9.6455	10.7193	10.8116
8	1.0149	2.6422	4.7778	5.9102	6.0364	5.8782
12	1.0144	2.3794	4.1310	4.3846	4.6392	3.6699
20	1.0135	1.8992	2.8842	2.6720	2.4262	2.4048
32	1.0121	1.4284	1.9171	1.5499	1.6342	1.5390

Architecture type: 4.

C\P	2	4	8	16	32	64
2	1.0156	2.9876	6.4516	12.8430	17.3265	20.6745
4	1.0154	2.8826	5.8190	10.0409	12.3837	11.9593
8	1.0149	2.6428	4.8437	6.7737	6.4271	7.0373
12	1.0144	2.3802	4.1763	4.4513	5.3318	4.7689
20	1.0135	1.9000	2.9750	2.6836	3.2979	3.0032
32	1.0121	1.4285	1.9769	1.8577	1.9857	1.9653

[Table 40]
[INTERFACE] Matrix: size=3516 nzero=81390 Density=0.012883%
[Generate] Diagonal to all elements in the column
[Generate] Two row elements in one node
[Generate] Time: mul_sub=11, div=6, two_op=20

PARASIM Simulation results
Matrix size=3516, Non-zero=81390, Density=0.012883
Network: Nodes=43357 Links=121230 Tasks=3
Timing: T_{all}=922875 T_{cpa}=18707 T_{local}=1
Best Speedup=49.33, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.4113	5.3612	11.1835
8	1.8673	3.9985	8.2682
20	1.1115	2.2654	4.6330
40	0.6633	1.3148	2.6708

Architecture type: 2.

C\P	4	8	16
4	2.3224	4.5178	4.4749
8	1.6127	2.5816	2.6347
16	1.0056	1.4422	1.4777

Architecture type: 3.

C\P	4	8	16
4	2.9308	5.8707	7.4698
8	2.6826	4.8546	4.9384
16	2.1440	3.5051	2.5287

Architecture type: 4.

C\P	4	8	16
4	2.9307	5.8726	8.2750
8	2.6820	4.8723	5.2744
16	2.1458	3.5875	3.0701

[Table 41]
[INTERFACE] Matrix: size=3516 nzero=81390 Density=0.012883%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 5
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=50

PARASIM Simulation results
Matrix size=3516, Non-zero=81390, Density=0.012883
Network: Nodes=21674 Links=99547 Tasks=4
Timing: T_{all}=900774 T_{cpa}=63127 T_{local}=1
Best Speedup=14.27, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.6193	5.8563	10.0935
8	2.0687	4.5170	8.0174
20	1.2672	2.6752	4.8095
40	0.7698	1.5922	2.8757

Architecture type: 2.

C\P	4	8	16
4	2.5337	4.2079	3.3064
8	1.8133	2.5667	2.2019
16	1.1402	1.3621	1.0935

Architecture type: 3.

C\P	4	8	16
4	3.1166	6.3340	7.0138
8	2.9790	5.7282	5.1697
16	2.7000	4.8590	3.3769

Architecture type: 4.

C\P	4	8	16
4	3.1166	6.3583	7.1831
8	2.9790	5.7407	6.0657
16	2.7000	4.8359	4.3179

[Table 42]
[INTERFACE] Matrix: size=3516 nzero=81390 Density=0.012883%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 5
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=50

PARASIM Simulation results
Matrix size=3516, Non-zero=81390, Density=0.012883
Network: Nodes=21378 Links=99251 Tasks=4
Timing: T_{all}=900303 T_{cpa}=7539 T_{local}=1
Best Speedup=119.42, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.4837	5.5380	11.6669
8	1.9952	4.2821	8.9072
20	1.2548	2.5456	5.2076
40	0.7753	1.5188	3.0767

Architecture type: 2.

C\P	4	8	16
4	2.4576	4.8835	5.8876
8	1.8358	3.0119	3.3410
16	1.1941	1.6623	1.7587

Architecture type: 3.

C\P	4	8	16
4	2.9337	6.3103	11.7282
8	2.8172	5.6445	8.5881
16	2.5667	4.6040	5.2221

Architecture type: 4.

C\P	4	8	16
4	2.9338	6.2849	12.0242
8	2.8146	5.6240	10.0448
16	2.5670	4.7536	6.5275

[Table 43]
[INTERFACE] Matrix: size=3516 nzero=81390 Density=0.012883%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 5
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=50

PARASIM Simulation results
Matrix size=3516, Non-zero=81390, Density=0.012883
Network: Nodes=21814 Links=99687 Tasks=4
Timing: T_{all}=900903 T_{cpa}=35739 T_{local}=1
Best Speedup=25.21, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.5445	5.6627	11.3129
8	2.0237	4.3666	8.7343
20	1.2527	2.5866	5.1811
40	0.7660	1.5401	3.0756

Architecture type: 2.

C\P	4	8	16
4	2.4885	4.6085	4.3364
8	1.7950	2.7612	2.6299
16	1.1459	1.5613	1.5365

Architecture type: 3.

C\P	4	8	16
4	3.0238	6.3205	6.7555
8	2.8887	5.6456	5.8317
16	2.6214	4.5448	4.2108

Architecture type: 4.

C\P	4	8	16
4	3.0239	6.3433	7.2707
8	2.8889	5.6817	6.9375
16	2.6217	4.6452	4.9499

[Table 44]
[INTERFACE] Matrix: size=3516 nzero=59376 Density=0.009322%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 5
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=50

PARASIM Simulation results
Matrix size=3516, Non-zero=59376, Density=0.009322
Network: Nodes=16881 Links=72740 Tasks=4
Timing: T_{all}=653697 T_{cpa}=32246 T_{local}=1
Best Speedup=20.27, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.5762	5.7907	11.7709
8	2.0485	4.4688	9.1019
20	1.2674	2.6493	5.3842
40	0.7747	1.5778	3.2017

Architecture type: 2.

C\P	4	8	16
4	2.5045	4.2287	4.2098
8	1.8220	2.5021	2.6790
16	1.1723	1.3526	1.4840

Architecture type: 3.

C\P	4	8	16
4	3.0406	6.4046	6.9531
8	2.8823	5.6332	5.3651
16	2.5862	4.5938	3.8102

Architecture type: 4.

C\P	4	8	16
4	3.0399	6.4362	7.5592
8	2.8882	5.7448	6.0306
16	2.5802	4.5995	4.7445

[Table 45]
[New Run] Matrix: size=3516 nzero=81390 Density=0.012883%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 10
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=100

PARASIM Simulation results
Matrix size=3516, Non-zero=81390, Density=0.012883
Network: Nodes=16967 Links=94840 Tasks=4
Timing: Tall=896241 Tcpa=113409 Tlocal=1
Best Speedup=7.90, CPA On

Architecture type: 2.

C\P	4	8	16
4	2.6894	3.2529	3.1961
8	1.9907	1.9199	2.4579
16	1.3076	1.0409	1.4322

Architecture type: 3.

C\P	4	8	16
4	3.1634	5.3793	4.5670
8	3.0512	4.8425	3.9648
16	2.8225	4.3474	3.5062

Architecture type: 4.

C\P	4	8	16
4	3.1634	5.3592	5.0302
8	3.0512	4.9270	4.5692
16	2.8225	4.3189	3.9120

[Table 46]
[INTERFACE] Matrix: size=3516 nzero=81390 Density=0.012883%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 10
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=100

PARASIM Simulation results
Matrix size=3516, Non-zero=81390, Density=0.012883
Network: Nodes=17166 Links=95039 Tasks=4
Timing: T_{all}=896350 T_{cpa}=20451 T_{local}=1
Best Speedup=43.83, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.5417	5.6640	11.9426
8	2.0548	4.4144	9.2053
20	1.3043	2.6522	5.4388
40	0.8106	1.5918	3.2318

Architecture type: 2.

C\P	4	8	16
4	2.4611	4.8277	5.1202
8	1.9636	3.3073	3.1530
16	1.2858	1.7318	1.6415

Architecture type: 3.

C\P	4	8	16
4	2.9798	6.5218	10.0573
8	2.8585	5.9962	8.1099
16	2.6410	5.0361	4.9478

Architecture type: 4.

C\P	4	8	16
4	2.9795	6.5247	12.0059
8	2.8587	6.0192	8.3160
16	2.6477	5.0859	6.5382

[Table 47]
[INTERFACE] Matrix: size=3516 nzero=81390 Density=0.012883%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 10
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=100

PARASIM Simulation results
Matrix size=3516, Non-zero=81390, Density=0.012883
Network: Nodes=17493 Links=95366 Tasks=4
Timing: Tall=897011 Tcpa=63261 Tlocal=1
Best Speedup=14.18, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.6530	5.8399	10.0642
8	2.1260	4.5657	7.9990
20	1.3303	2.7547	4.8787
40	0.8189	1.6577	2.9516

Architecture type: 2.

C\P	4	8	16
4	2.5557	4.1105	3.5564
8	1.8993	2.5906	2.5008
16	1.2140	1.5234	1.4932

Architecture type: 3.

C\P	4	8	16
4	3.0552	6.1627	4.8726
8	2.9488	5.5189	4.2520
16	2.7037	4.7428	3.6049

Architecture type: 4.

C\P	4	8	16
4	3.0547	6.1293	4.7875
8	2.9450	5.5829	4.6256
16	2.7051	4.7264	4.0355

[Table 48]
[New Run] Matrix: size=9289 nzero=250689 Density=0.005703%
[Generate] Diagonal to all elements in the column
[Generate] Two row elements in one node
[Generate] Time: mul_sub=11, div=6, two_op=20

PARASIM Simulation results
Matrix size=9289, Non-zero=250689, Density=0.005703
Network: Nodes=132142 Links=373541 Tasks=3
Timing: T_{all}=2847582 T_{cpa}=37178 T_{local}=1
Best Speedup=76.59, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.3961	5.3181	11.1339
8	1.8599	3.9732	8.2197
20	1.1113	2.2562	4.5966
40	0.6646	1.3112	2.6490

Architecture type: 2.

C\P	4	8	16
4	2.3067	4.4585	4.3972
8	1.6078	2.5846	2.5283
16	1.0014	1.3807	1.3567

Architecture type: 3.

C\P	4	8	16
4	2.9133	5.8310	7.4437
8	2.6683	4.8723	4.5511
16	2.1285	3.5476	2.4638

Architecture type: 4.

C\P	4	8	16
4	2.9133	5.8500	8.1630
8	2.6683	4.8693	5.1993
16	2.1285	3.5519	2.8326

[Table 49]
[INTERFACE] Matrix: size=9289 nzero=250689 Density=0.005703%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 5
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=50

PARASIM Simulation results
Matrix size=9289, Non-zero=250689, Density=0.005703
Network: Nodes=64495 Links=305894 Tasks=4
Timing: T_{all}=2778816 T_{cpa}=70894 T_{local}=1
Best Speedup=39.20, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.5104	5.5867	11.5205
8	2.0022	4.3164	8.8373
20	1.2447	2.5615	5.1943
40	0.7631	1.5259	3.0750

Architecture type: 2.

C\P	4	8	16
4	2.4248	4.4924	4.1400
8	1.7367	2.6772	2.4835
16	1.1056	1.4675	1.4130

Architecture type: 3.

C\P	4	8	16
4	2.9737	6.2643	8.0468
8	2.8525	5.6249	5.7429
16	2.6008	4.6749	3.4701

Architecture type: 4.

C\P	4	8	16
4	2.9737	6.2258	8.6387
8	2.8525	5.6514	7.2852
16	2.6008	4.6860	5.3369

[Table 50]
[INTERFACE] Matrix: size=9289 nzero=250689 Density=0.005703%
[Generate] Diagonal to all elements in the column
[Generate] Elements in one node are 10
[Generate] Time: mul_sub=11, div=6, two_op=20, Multi=100

PARASIM Simulation results
Matrix size=9289, Non-zero=250689, Density=0.005703
Network: Nodes=50462 Links=291861 Tasks=4
Timing: T_{all}=2765902 T_{cpa}=120342 T_{local}=1
Best Speedup=22.98, CPA On

Architecture type: 1.

C\P	4	8	16
4	2.5771	5.7481	11.6770
8	2.0699	4.4906	9.0962
20	1.2997	2.7054	5.4477
40	0.8019	1.6255	3.2586

Architecture type: 2.

C\P	4	8	16
4	2.4805	4.1030	3.5850
8	1.8002	2.5131	2.2779
16	1.1524	1.4542	1.2558

Architecture type: 3.

C\P	4	8	16
4	3.0156	6.2295	6.2928
8	2.9047	5.7813	5.2356
16	2.6899	4.8745	3.6295

Architecture type: 4.

C\P	4	8	16
4	3.0156	6.1567	5.5913
8	2.9047	5.8092	5.9142
16	2.6899	4.8861	4.6470

Appendix B: **Program Listings**

This appendix includes the main routines in the PARASIM program. They are grouped as follows:

1- Network generation routines.
2- Scheduling program.
3- Multiprocessor simulation routines.
4- Output and display routines.
5- Matrix operations routines.

The following is a list of the routine names, source file name and the function of each procedure:

Network Generation Routines

Name: interface():	Source File: Parasim1.c

This routine is the entry point from both the main() and the Mega programs. It passes all the matrix data needed for the simulation of the execution of the network.
it also opens and closes the files used in saving the program acitivity and results.

Name: create_task():	Source File: Parasim1.c

This routine will reserve the memory for a task structure.

Name: c_link():	Source File: Parasim3.c

This routine will reserve the memory for a link structure.

Name: ln_1():	Source File: Parasim3.c

This routine will create a link structure and joins the source and destination nodes. This will represent a data transfer from the node to the other.

Name: new_network_generate():	Source File: Parasim3.c

This routine will generate the network details form the matrix information.

Scheduling Routines

Name: is_input_ready():	Source File: Parasim1.c

This routine will return the status of the node. This returns the satus of all the data input links.

Name: is_output_ready():	Source File: Parasim1.c

This routine will check if all the input links have their data ready. This would enable the node to be executed.

Name: all_inputs_not_avail():	Source File: Parasim1.c

This routine places all nodes that are connected only to the head_node into the line_list. This would enable the start of execution for the reverse_pass() routine.

Name: win_put():	Source File: Parasim1.c

This routine will place the node in the line_list.

Name: win_remove():	Source File: Parasim1.c

This routine will remove the node from the line_list. If the node is not in the line_list it will be reported to the user.

Name: is_node_critical():	Source File: Parasim1.c

This routine will test if the given node is critical (i.e., resides on the critical path).

Name: put_all_nodes_of_input():	Source File: Parasim1.c

This routine will place all nodes that are recieve inputs only from the head_node into the line_list.

Name: get_number_of_outs():	Source File: Parasim1.c

This routine computes the number of outputs leaving the given node.

Name: number_of_outs():	Source File: Parasim2.c

This routine will return the number of outputs for the given node.

Name: get_next_link():	Source File: Parasim1.c

This routine will return next available link, which has a ready input available and the output is not yet ready. This link would need the initiation of a communication routine to transfer the data.

Name: new_node_executed():	Source File: Parasim2.c

This routine will mark the node as executed. It will compute the earliest stat times. It will place all the nodes that recieve data into the line_list.

Name: new1_node_executed():	Source File: Parasim2.c

This routine will increase the processor time by the task time, assigns processor time to the node and all its output links.

Name: new_fwd_pass():	Source File: Parasim2.c

This routine will compute the earliest start time for all the nodes.

Name: init1_rev_pass():	Source File: Parasim2.c

This routine will scan the complete network to identify the nodes that are connected to the head_node.

Name: new_node_dexecuted():	Source File: Parasim2.c

This node will establish the latest linkt of the given node. and compute the critical path time and the number of critical nodes.

Name: new_rev_pass()	Source File: Parasim2.c

This routine computs the latest start time for all the nodes.

Multiprocessor Simulation Routines

Name: win_reset_variables():	Source File: Parasim1.c

For all the nodes of the network all variables are set to the initial state.

Name: reset_variables():	Source File: Parasim2.c

For all the nodes of the network all variables are set to the initial state.

Name: reset_procs():	Source File: Parasim2.c

This routine will reset all the processors to the initial state. The bus status information is are reset.

Name: bus_time():	Source File: Parasim2.c

This routines computes and returns the time of the multiprocessor bus.

Name: p_time():	Source File: Parasim2.c

This routine will return the current time of the given processor. This time will consist of all active and idle cycles.

Name: least_link():	Source File: Parasim2.c

This routine will return the link that has least communication time.

Name: bus_server():	Source File: Parasim2.c

This routine will simulate the different architectures. It will also allow the simulation of bus contention and special hardware features.

Name: new_next_node_proc():	Source File: Parasim2.c

This routine will return next node available for execution.

Name: simulation():	Source File: Parasim2.c

This is main routine for the simuation of the execution of the network.

Name: total_procs():	Source File: Parasim2.c

This routine groups the activity of the processor into: active, communicate and idle. It will also print the resulting values into the results file.

Output and Display Routines

Name: print_network():	Source File: Parasim1.c

This routine will print out all the nodes and their connections along with times associated with them.

Name: ps_mat():	Source File: Parasim1.c

This routine will convert the matrix data into a form to be printed by the Lotus Manuscript wordprocessor.

Name: print_critical():	Source File: Parasim1.c

This routine will print a list of critical nodes information.

Name: comments():	Source File: Parasim1.c

This routine will place a number of comments lines into the results' files.

Name: print_heading():	Source File: Parasim1.c

This routine prints the heading of the program which includes the program's name and version, date, time and remaining memory details.

Name: print_header():	Source File: Parasim2.c

Will print the table and graph heading information.

Name: print_procs():	Source File: Parasim.c

This routine prints information about the processors' activities.

Name: date_time():	Source File: Parasim3.c

This routine will print the system date and time.

Name: network_info():	Source File: Parasim3.c

This routine will display all the information regarding the network structure and organization.

Name: bus_info():	Source File: Parasim3.c

This routine will display the multiprocessor bus information.

Name: print_elements():	Source File: Parasim3.c

This routine will save the matrix information into (X,Y) format using Postscript language.

Name: mat_print_1():	Source File: Parasim3.c

This routine will display the top for each matrix entry.

Name: mat_print_1():	Source File: Parasim3.c

This routine will display the top for each matrix entry.

Name: mat_print_2():	Source File: Parasim3.c

This routine will display the element's column value and the top for each matrix entry.

Menu and Selection Routines

Name: menu():	Source File: Parasim2.c

This is the main menu for Parasim.

Name: menu_1():	Source File: Parasim2.c

This routine selects the architecture and the output table to be used for the simulation.

Name: menu_2():	Source File: Parasim3.c

This routine displays and selects the Utilities subprograms.

Results and Test routines

Name: test_broadcast():	Source File: Parasim1.c

This routine produces a graph which depicts the effect of broadcast on the execution of the network.

Name: form_00():	Source File: Parasim2.c

This routine will produce a line showing the effect of distributed and shared memory models. The detail processors' activities is printed.

Name: test_all():	Source File: Parasim2.c

This routine tests all the architectures and report it to the useer.

Name: multi_simula_1():	Source File: Parasim2.c

This routine will produce a table of results for the output file.

Name: multi_simula_2():	Source File: Parasim2.c

This routine will produce a table of results for the output file.

Name: multi_simula_3():	Source File: Parasim2.c

This routine will produce a table of results for the output file.

Name: comm_matrix():	Source File: Parasim3.c

This routine will print the number of data links between each pair of processing nodes.

Matrix operations routines

Name: remove_element():	Source File: Parasim1.c

This routine will remove one element from the matrix. It will adjust the diagonal[] and column_no[] arrays that describe the matrix.

Name: convert_matrix():	Source File: Parasim3.c

This routine is used to generate the Top array.

Name: data_consist():	Source File: Parasim3.c

This routine checks that the data structures of the network are consistent with the arrays passed from the calling program.

Name: network_check():	Source File: Parasim3.c

This routine is used to identify the nodes latest times.

Name: save_data():	Source File: Parasim3.c

This routine will save the arrays representing the matrix structure into a file. Also inserted are extra checking information.

Name: matrix_generate():	Source File: Parasim3.c

This routine will create the matrix from random number generator.

Name: element():	Source File: Parasim3.c

This routine will return the element number of the matrix entry.

Name: read_data():	Source File: Parasim3.c

This routine will read from a file the complete matrix information. Option to select the version of the data file.